Table of Contents

CORE JAVA

JAVA

Java is a general-purpose programming language intended to let programmers ***Write Once, Run Anywhere (WORA)***. This means that compiled Java code can run on all platforms that support Java without the need to recompile.

Why java

- **OBJECT-ORIENTED**: In Java, everything is an object. Java can be easily extended since it is based on the object model.
- **PLATFORM INDEPENDENT**: Unlike many other programming languages, including C and C++, when Java is compiled, it is not compiled into a platform-specific machine but rather into platform-independent byte code. This byte code is distributed over the web and interpreted by the virtual machine (JVM) on whichever platform it is being run on.
- **SIMPLE**: Java is designed to be easy to learn. If you understand the basic concept of OOP Java, it would be easy to master.
- **SECURE**: With Java's secure feature, it enables the development of virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.
- **ARCHITECTURE-NEUTRAL**: The Java compiler generates an architecture-neutral object file format, which makes the compiled code executable on many processors with the presence of the Java runtime system.
- **PORTABLE**: Being architecture-neutral and having no implementation-dependent aspects of the specification makes Java portable. Compiler in Java is written in ANSI C with a clean portability boundary, which is a POSIX subset.
- **ROBUST**: Java tries to eliminate error-prone situations by emphasizing mainly compile-time error checking and runtime checking.
- **MULTITHREADED**: With Java's multithreaded feature, it is possible to write programs that can perform many tasks simultaneously. This design feature allows the developers to construct interactive applications that can run smoothly.
- **INTERPRETED**: Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light-weight process.
- **HIGH PERFORMANCE**: With the use of just-in-Time compilers, Java enables high performance.
- **DISTRIBUTED**: Java is designed for the distributed environment of the internet.
- **DYNAMIC**: Java is more dynamic than C or C++ because it adapts to changing environments. Java programs can carry an extensive amount of run-time information that can be used to verify and resolve accesses to objects in run-time.

Java History Version

No.	Version	Date	Description
1	JDK Beta	1995	Initial Draft version
2	JDK 1.0	23 Jan 1996	A stable variant JDK 1.0.2 was termed as JDK 1
3	JDK 1.1	19 Feb 1997	Major features like JavaBeans, RMI, JDBC, inner classes were added in this release.
4	JDK 1.2	8 Dec 1998	Swing, JIT Compiler, Java Modules, Collections were introduced to JAVA and this release was a great success.
5	JDK 1.3	8 May 2000	HotSpot JVM, JNDI, JPDA, JavaSound and support for Synthetic proxy classes were added.
6	JDK 1.4	6 Feb 2002	Image I/O API to create/read JPEG/PNG image were added. Integrated XML parser and XSLT processor (JAXP) and Preferences API were other important updates.
7	JDK 1.5 or J2SE 5	30 Sep 2004	Various new features were added to the language like foreach, var-args, generics etc.
8	JAVA SE 6	11Dec 2006	1. notation was dropped to SE and upgrades done to JAXB 2.0, JSR 269 support and JDBC 4.0 support added.
9	JAVA SE 7	7 Jul 2011	Support for dynamic languages added to JVM. Another enhancement included string in switch case, compressed 64 bit pointers etc.
10	JAVA SE 8	18 Mar 2014	Support for functional programming added. Lambda expressions streams, default methods, new date-time APIs introduced.
11	JAVA SE 9	21 Sep 2017	Module system introduced which can be applied to JVM platform.

12	JAVA SE 10	20 Mar 2018	Unicode language-tag extensions added. Root certificates, threadlocal handshakes, support for heap allocation on alternate memory devices etc were introduced.
13	JAVA SE 11	5 Sep 2018	Dynamic class-file constants, Epsilon a no-op garbage collector, local-variable support in lambda parameters, Low-overhead heap profiling support added.
14	JAVA SE 12	19 Mar 2019	Experimental Garbage Collector,Shenandoah: A Low-Pause-Time Garbage Collector, Microbenchmark Suite, JVM Constants API added.
15	JAVA SE 13	17 Sep 2019	Feature added - Text Blocks (Multiline strings), Enhanced Thread-local handshakes.
16	JAVA SE 14	17 Mar 2020	Feature added - Records, a new class type for modelling, Pattern Matching for instanceof, Intuitive NullPointerException handling.
17	JAVA SE 15	15 Sep 2020	Feature added - Sealed Classes, Hidden Classes, Foreign Function and Memory API (Incubator).
18	JAVA SE 16	16 Mar 2021	Feature added as preview - Records, Pattern Matching for switch, Unix Domain Socket Channel (Incubator) etc.
19	JAVA SE 17	14 Sep 2021	Feature added as finalized - Sealed Classes, Pattern Matching for instanceof, Strong encapsulation of JDK internals by default. New macOS rendering pipeline etc.
20	JAVA SE 18	22 Mar 2022	Feature added - UTF-8 by Default, Code Snippets in Java API Documentation, Vector API (Third incubator), Foreign Function, Memory API (Second Incubator) etc.

21	JAVA SE 19	20 Sep 2022	Feature added - Record pattern, Vector API (Fourth incubator), Structured Concurrency (Incubator) etc.
22	JAVA SE 20	21 Mar 2023	Feature added - Scoped Values (Incubator), Record Patterns (Second Preview), Pattern Matching for switch (Fourth Preview),Foreign Function & Memory API (Second Preview) etc.
22	JAVA SE 21	19 Sep 2023	Feature added - String Templates (Preview), Sequenced Collections, Generational ZGC, Record Patterns, Pattern Matching for switch etc.
23	Java SE 22	19 Mar 2024	Feature added - Region Pinning for G1 garbage collector, foreign functions and memory APIs , multi-file source code programs support, string templates, vector apis (seventh incubator), unnamed variables, patterns, stream gatherers (first preview) etc.
24	Java SE 23	17 Sep 2024	Feature added - Primitive types in patterns, class file APIs, vector APIs (Eighth incubator), stream gatherers (second preview), ZDC, generation mode by default etc.

COMPILER VS INTERPRETER

Compiler

A compiler is a program that translates the entire source code of a programming language (like Java, C, or C++) into machine code (binary instructions) or an intermediate code (like Java bytecode) before the program is run. This translation happens all at once, and the output is usually a separate executable file or a set of instructions that can be run by a virtual machine.

How it works: The compiler reads the whole source code, checks for errors, and converts it into machine code. If there are errors, it reports them all at once after scanning the code.

Execution: After compilation, the resulting machine code can be executed multiple times without needing the source code or the compiler.

Examples: Java compiler (javac), C compiler (gcc), C++ compiler (g++).

Interpreter

An interpreter is a program that reads and executes the source code line by line, translating each statement into machine code or intermediate code as the program runs. It does not produce a separate executable file; instead, it directly executes the instructions.

How it works: The interpreter reads one line or statement at a time, translates it, and executes it immediately. If it encounters an error, it stops execution at that point.

Execution: The source code must be present every time the program runs, and the interpreter is needed to execute it.

Examples: Python interpreter (python), JavaScript engines in browsers, Ruby interpreter.

Key Differences

Translation:

Compiler: Translates the whole program before execution.

Interpreter: Translates and executes line by line during execution.

Speed:

Compiler: Faster execution after compilation, since the code is already translated.

Interpreter: Slower, as translation happens during execution.

Error Handling:

Compiler: Reports all errors after scanning the entire code.

Interpreter: Stops at the first error encountered during execution.

Output:

Compiler: Generates an independent executable or intermediate code.

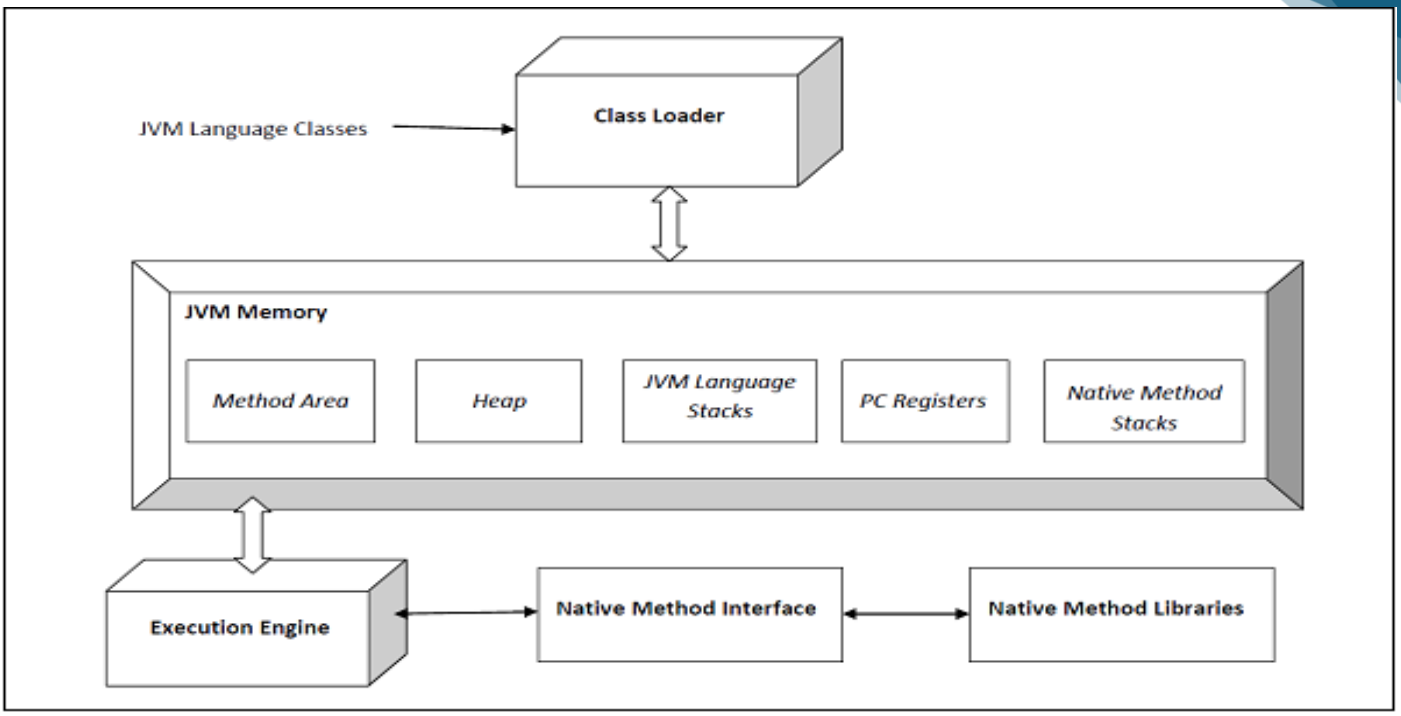
Interpreter: No separate output file; executes directly.

Java uses both

- The Java compiler (javac) translates Java source code into bytecode (.class files).
- The Java Virtual Machine (JVM) acts as an interpreter (and sometimes a Just-In-Time compiler) to execute the bytecode.

JVM (Java Virtual Machine)

The Java Virtual Machine (JVM) is a virtual engine that runs Java bytecode. It acts as a bridge between your compiled Java code and the underlying hardware. When you write Java code and compile it, the compiler generates bytecode (a .class file). This bytecode is not directly executed by the operating system—it's executed by the JVM, which interprets or compiles it into native machine code.



Example Flow

- You write Java code → HelloWorld.java
- Compile it → javac HelloWorld.java → creates HelloWorld.class
- Run it → java HelloWorld
- JVM loads HelloWorld.class, verifies it, and executes it.

Components of JVM (Java Virtual Machine) Architecture

The following are the main components of JVM (Java Virtual Machine) architecture:

1. Class Loader

The JVM manages the process of loading, linking and initializing classes and interfaces in a dynamic manner. During the loading process, the JVM finds the binary representation of a class and creates it.

During the linking process, the loaded classes are combined into the run-time state of the JVM so that they can be executed during the initialization phase. The JVM basically uses the symbol table stored in the run-time constant pool for the linking process. Initialization consists of actually executing the linked classes.

The following are the types of class loaders:

BootStrap class loader: This class loader is on the top of the class loader hierarchy. It loads the standard JDK classes in the JRE's lib directory.

Extension class loader: This class loader is in the middle of the class loader hierarchy and is the immediate child of the bootstrap class loader and loads the classes in the JRE's lib\ext directory.

Application class loader: This class loader is at the bottom of the class loader hierarchy and is the immediate child of the application class loader. It loads the jars and classes specified by the CLASSPATH ENV variable.

2. Linking and Initialization

The linking process consists of the following three steps –

Verification – This is done by the Bytecode verifier to ensure that the generated .class files (the Bytecode) are valid. If not, an error is thrown and the linking process comes to a halt.

Preparation – Memory is allocated to all static variables of a class and they are initialized with the default values.

Resolution – All symbolic memory references are replaced with the original references. To accomplish this, the symbol table in the run-time constant memory of the method area of the class is used.

Initialization is the final phase of the class-loading process. Static variables are assigned original values and static blocks are executed.

3. Runtime Data Areas

The JVM spec defines certain run-time data areas that are needed during the execution of the program. Some of them are created while the JVM starts up. Others are local to threads and are created only when a thread is created (and destroyed when the thread is destroyed). These are listed below –

PC (Program Counter) Register

It is local to each thread and contains the address of the JVM instruction that the thread is currently executing.

Stack

It is local to each thread and stores parameters, local variables and return addresses during method calls. A StackOverflow error can occur if a thread demands more stack space than is permitted. If the stack is dynamically expandable, it can still throw OutOfMemoryError.

Heap

It is shared among all the threads and contains objects, classes' metadata, arrays, etc., that are created during run-time. It is created when the JVM starts and is destroyed when the JVM shuts down. You can control the amount of heap your JVM demands from the OS using certain flags (more on this later). Care has to be taken not to demand too less or too much of the memory, as it has important performance implications. Further, the GC manages this space and continually removes dead objects to free up the space.

Method Area

This run-time area is common to all threads and is created when the JVM starts up. It stores per-class structures such as the constant pool (more on this later), the code for constructors and methods, method data, etc. The JLS does not specify if this area needs to be garbage collected, and hence, implementations of the JVM may choose to ignore GC. Further, this may or may not expand as per the application's needs. The JLS does not mandate anything regarding this.

Run-Time Constant Pool

The JVM maintains a per-class/per-type data structure that acts as the symbol table (one of its many roles) while linking the loaded classes.

Native Method Stacks

When a thread invokes a native method, it enters a new world in which the structures and security restrictions of the Java virtual machine no longer hamper its freedom. A native method can likely access the runtime data areas of the virtual machine (it depends upon the native method interface), but can also do anything else it wants.

4. Execution Engine

The execution engine is responsible for executing the bytecode, it has three different components:

Garbage Collection

The JVM manages the entire lifecycle of objects in Java. Once an object is created, the developer need not worry about it anymore. In case the object becomes dead (that is, there is no reference to it anymore), it is ejected from the heap by the GC using one of the many algorithms serial GC, CMS, G1, etc.

During the GC process, objects are moved in memory. Hence, those objects are not usable while the process is going on. The entire application has to be stopped for the duration of the process. Such pauses are called 'stop-the-world' pauses and are a huge overhead. GC algorithms aim primarily to reduce this time.

Interpreter

The interpreter Interprets the bytecode. It interprets the code fast but it's slow in execution.

JIT Complier

The JIT stands for Just-In-Time. The JIT compiler is a main part of the Java runtime environment and it compiles bytecodes to machine code at runtime.

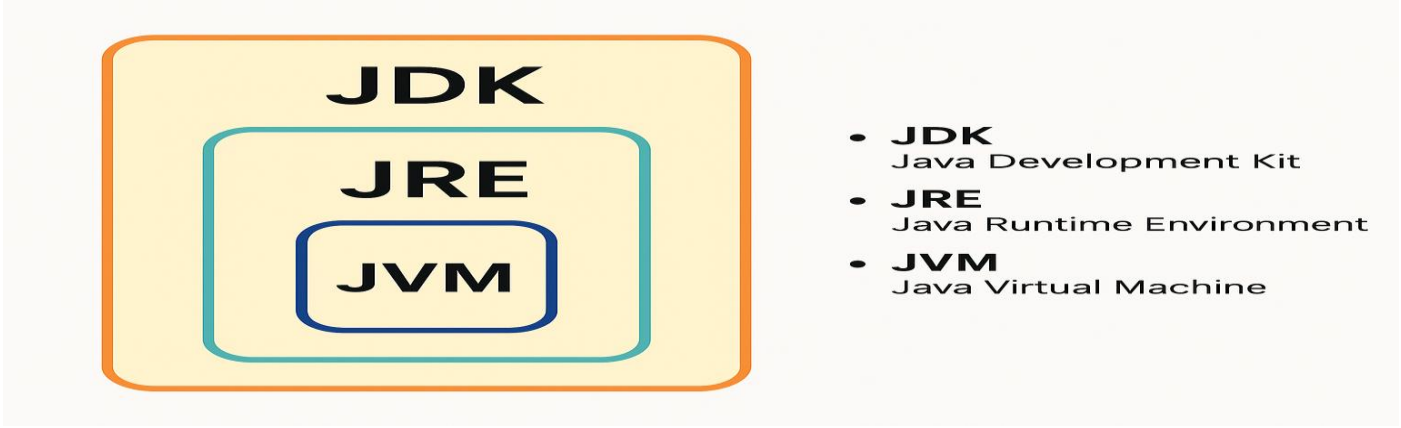
5. Java Native Interface (JNI)

Java Native Interface (JNI) interacts with the native method libraries which are essential for the execution.

6. Native Method Libraries

Native method libraries are the collection of C and C++ libraries (native libraries) which are essential for the execution.

JDK, JRE, and JVM



1. JVM (Java Virtual Machine)

The JVM is the engine that runs Java bytecode. It provides a runtime environment that interprets or compiles .class files (compiled Java code) into machine code specific to the host system. JVM is platform-dependent, meaning each operating system has its own implementation of JVM.

🔑 Key Points:

- ❖ Executes Java bytecode.
- ❖ Handles memory management (heap, stack).
- ❖ Performs garbage collection.
- ❖ Provides security and portability.

📌 Notes:

- ❖ JVM is part of both JRE and JDK.
- ❖ It does not include development tools like compilers.

2. JRE (Java Runtime Environment)

The JRE is a package that contains the JVM along with the core libraries and other components required to run Java applications. It is meant for users who want to run Java programs but not develop them.

🔑 Key Points:

- ❖ Includes JVM + Java class libraries + supporting files.
- ❖ Does not include development tools like javac (Java compiler).
- ❖ Ideal for end-users who only need to run Java applications.

📌 Notes:

- ❖ If you're only running Java apps (not writing code), JRE is sufficient.
- ❖ JRE is platform-specific.

3. JDK (Java Development Kit)

The JDK is the full-featured software development kit for Java. It includes everything in the JRE plus tools needed for Java development, such as the compiler (javac), debugger (jdb), and other utilities.

🔑 Key Points:

- ❖ Includes JRE + development tools.
- ❖ Required for writing, compiling, and debugging Java programs.
- ❖ Contains tools like javac, Javadoc, jar, etc.

📌 Notes:

- ❖ Developers need JDK to build Java applications.
- ❖ JDK is also platform specific.

Comparison Table

Feature	JVM	JRE	JDK
Runs Java Code	✅	✅	✅
Includes JVM	✅	✅	✅
Includes Libraries	❌	✅	✅
Includes Compiler	❌	❌	✅
For Development	❌	❌	✅
For Execution	✅	✅	✅

Java - Basic Syntax

```
1  public class MyFirstJavaProgram {
2     public static void main(String []args) {
3        System.out.println("Hello World");
4     }
5  }
```

1) public

Keyword: An access modifier.

Meaning: Makes the class (or method) accessible from anywhere—other classes, other packages.

public class MyFirstJavaProgram → The class is accessible globally.

public static void main → The JVM must be able to call main from outside; that's why main is public.

2) class

Keyword: Defines a class, which is a blueprint for objects or a container for code (fields, methods).

class MyFirstJavaProgram declares a class named MyFirstJavaProgram.

File name rule: When a class is public, the file name must match the class name exactly (case-sensitive) with .java extension:

File must be: MyFirstJavaProgram.java

3) MyFirstJavaProgram

Identifier: The name of the class.

Naming convention: Pascal Case (Each word capitalized). Java convention recommends class names starting with uppercase letters.

Role: Containers the main method; serves as the entry point class for this simple program.

4) { and }

Braces: Define blocks.

Outer braces: Enclose everything that belongs to the class MyFirstJavaProgram.

Inner braces: Enclose the body of the main method.

5) public static void main(String []args)

public

Accessible to the JVM from anywhere so it can invoke your program's entry point.

static

Belongs to the class, not to an instance. The JVM can call it without creating an object of MyFirstJavaProgram.

void

The method does not return any value.

main

The method name recognized by the JVM as the entry point for console applications.

(String []args)

Parameter: One argument named args, which is an array of Strings.

It contains command-line arguments passed when you run the program, e.g., java MyFirstJavaProgram hello world → args[0] = "hello", args[1] = "world".

Style note: Both String[] args and String []args are valid; the former is more common.

6) System.out.println("Hello World");

This single line prints to the console and then moves to a new line.

Let's dissect it:

System

A final class in java.lang (automatically imported) that provides system-level utilities.

out

A public static field inside System. Its type is PrintStream. It represents the standard output stream (usually your terminal/console).

println(...)

A method of PrintStream that prints the argument followed by a newline (\n). If you used print(...) instead, it would not add a newline.

"Hello World"

A String literal. Java strings are objects of type String. The content is Unicode, so you can print any language or emoji too.

A commented version for clarity

```java
// A public class; file name must be MyFirstJavaProgram.java
public class MyFirstJavaProgram {

    // Program entry point: public (JVM can access), static (no object needed), void (no return)
    // args: command-line arguments
    public static void main(String[] args) {
        // Print to standard output and add a newline
        System.out.println("Hello World");
    }
}
```

Variable Types

1. Local Variables

- Local variables are declared inside methods, constructors, or blocks, and they exist only during the execution of that block.
- They are created when the method or block starts and destroyed once it ends, making them temporary in nature.
- These variables are stored in stack memory, which is fast and suitable for short-lived data.
- Access modifiers cannot be applied to local variables.
- They do not have default values, so you must initialize them before using them.
- Their scope is limited to the method or block where they are declared, ensuring they are not accessible outside.

2. Instance Variables

- Instance variables are declared inside a class but outside any method, constructor, or block.
- They represent the state of an object and are created when an object is instantiated using the new keyword.
- These variables are stored in heap memory and remain available as long as the object exists.
- Unlike local variables, instance variables have default values:
    - Numbers → 0
    - Boolean → false
    - Object references → null
- They can have access modifiers, and it is recommended to make them private for encapsulation.
- Instance variables can be accessed directly within the class or through an object reference outside the class.
- They are useful for data that needs to persist across multiple methods within the same object.

3. Class (Static) Variables

- Class variables, also called static variables, are declared using the static keyword inside a class but outside any method or block.
- There is only one copy of a static variable per class, shared by all objects of that class.
- These variables are stored in static memory and exist for the entire duration of the program.
- Static variables also have default values similar to instance variables.
- They are commonly used for constants, which are declared as public static final and typically written in UPPER_CASE.
- Static variables can be accessed using the class name rather than an object reference.
- They are ideal for data that should remain consistent across all instances, such as configuration values or counters.

JAVA DATA TYPES

In Java, data types define the kind of values a variable can hold and how much memory it occupies. They are divided into two main categories: Primitive Data Types and Reference (Object) Data Types.

1. Primitive Data Types

Primitive data types are predefined by Java and represent simple values. There are eight types, each serving a specific purpose:

byte

- 8-bit signed integer, range: -128 to 127
- Default value: 0
- Used to save memory in large arrays.
- Example: byte a = 100;

short

- 16-bit signed integer, range: -32,768 to 32,767
- Default value: 0
- Example: short s = 10000;

int

- 32-bit signed integer, range: -2,147,483,648 to 2,147,483,647
- Default value: 0
- Most commonly used for integers.
- Example: int a = 100000;

long

- 64-bit signed integer, range: very large
- Default value: 0L
- Used when int range is insufficient.
- Example: long a = 100000L;

float

- 32-bit single-precision floating-point
- Default value: 0.0f
- Not suitable for precise values like currency.
- Example: float f1 = 234.5f;

double

- 64-bit double-precision floating-point
- Default value: 0.0d
- Default choice for decimal values.
- Example: double d1 = 123.4;

boolean

- Represents true or false
- Default value: false

- Example: boolean one = true;

char

- 16-bit Unicode character, range: '\u0000' to '\uffff'
- Default value: '\u0000'
- Example: char letterA = 'A';

Key Points:

- ❖ Primitive types store actual values (not objects).
- ❖ They have fixed size, default values, and are efficient for simple data.

2. Reference (Object) Data Types

Reference data types store the memory address of objects rather than the actual data. They are not predefined like primitives but created using classes. Examples include:

String → Represents sequences of characters.

Arrays → Stores multiple values of the same type.

Classes → User-defined types with variables and methods.

Interfaces → Abstract types specifying method sets.

Key Points:

- ❖ Reference types can have methods and properties.
- ❖ Default value is null.
- ❖ They allow complex data structures and object-oriented programming.

Java Type Casting

Type casting in Java refers to converting one data type into another, either automatically by the compiler or manually by the programmer. It ensures compatibility between different data types during operations. There are two main types:

1. Widening Type Casting (Implicit Conversion)

- ❖ Widening type casting happens when a smaller data type is converted into a larger data type automatically by the compiler. This is safe because the larger type can store all values of the smaller type without data loss.
- ❖ The conversion follows a specific hierarchy:
- ❖ byte → short → char → int → long → float → double
- ❖ This process is common in arithmetic operations involving mixed types. For example, adding an int and a double results in a double because the int is implicitly converted.

Example:

```
1  int num1 = 5004;
2  double num2 = 2.5;
3  double sum = num1 + num2; // int converted to double automatically
4  System.out.println("Sum: " + sum); // Output: 5006.5
5
```

Type Conversion Error

- A type conversion error occurs when you try to assign a larger data type to a smaller data type without explicit casting. The compiler prevents this to avoid data loss.
- For example:

```
int sum = num1 + num2; // ERROR: cannot convert double to int
```

- This rule ensures type safety in Java.

2. Narrowing Type Casting (Explicit Conversion)

- Narrowing type casting occurs when a larger data type is converted into a smaller data type manually by the programmer using a cast operator. This is necessary because the compiler does not perform this conversion automatically due to the risk of data loss.
- The syntax for explicit casting is:

```
int convertedInt = (int) doubleValue;
```

- This is useful when you intentionally need to store a large type value in a smaller type variable.
- Example:

```
int num = 5004;
double doubleNum = (double) num; // int to double
int convertedInt = (int) doubleNum; // double to int
System.out.println("Double: " + doubleNum); // 5004.0
System.out.println("Int: " + convertedInt); // 5004
```

Summary

- Widening (Implicit): Automatic, safe, no data loss, smaller → larger type.
- Narrowing (Explicit): Manual, may cause data loss, larger → smaller type.

User Input in Java

- Java provides the Scanner class from the java.util package to read input from the user.
- The Scanner class offers multiple methods to read different types of data such as integers, floating-point numbers, strings, and booleans.
- It is widely used because it is simple and supports various input types.

Steps to Use Scanner for User Input

Import the Scanner Class

To use the Scanner class, you must first import it into your program.

```
import java.util.Scanner;
```

Create a Scanner Object

After importing, create an object of Scanner to access its methods.

```
1  Scanner sc = new Scanner(System.in);
```

Read User Input

Use appropriate methods like nextInt(), nextFloat(), or nextLine() to read input.

```
1  int age = sc.nextInt();
```

Common Scanner Methods

- nextInt() → Reads an integer value.
- nextFloat() → Reads a float value.
- nextDouble() → Reads a double value.
- nextBoolean() → Reads a boolean value (true or false).
- next() → Reads a single word (stops at whitespace).
- nextLine() → Reads an entire line including spaces.

Operators

- Java operators are symbols that perform operations on variables and values, enabling tasks like arithmetic calculations, comparisons, logical decisions, and bit-level manipulations.
- They are essential for building expressions and controlling program flow.
- Operators in Java are categorized into Arithmetic, Assignment, Relational, Logical, Bitwise, Miscellaneous, and Operator Precedence.

1. Arithmetic Operators

- Arithmetic operators are used for basic mathematical operations such as addition, subtraction, multiplication, division, and modulus.
- They also include increment (++) and decrement (--) operators for increasing or decreasing values by one.

2. Assignment Operators

- Assignment operators assign values to variables and allow shorthand operations for addition, subtraction, multiplication, and more.
- For example, a += 5 adds 5 to a and assigns the result back to a.

3. Relational Operators

- Relational operators compare two values and return a boolean result (true or false).
- Common operators include ==, !=, >, <, >=, and <=.
- These are widely used in conditional statements and loops.

4. Logical Operators

- Logical operators perform logical operations on Boolean values.
- They are commonly used in decision-making constructs like if statements.

- Operators include:
  - && (AND) → true if both conditions are true.
  - || (OR) → true if at least one condition is true.
  - ! (NOT) → reverses the logical state.

5. Bitwise Operators

- Bitwise operators work on binary representations of integers and perform operations at the bit level.
- Common operators:

  & (AND), **|** (OR), **^** (XOR), **~** (NOT), **<<** (Left Shift), **>>** (Right Shift), **>>>** (Unsigned Right Shift).

6. Miscellaneous Operators

1. Ternary Operator (?:)

Used for conditional expressions:

```
1  int a = 10;
2  int b = (a == 10) ? 20 : 30; // b = 20
```

2. instanceof Operator

Checks if an object is an instance of a class:

Purpose of the instanceof Operator

The **INSTANCEOF** operator is utilised to determine whether a particular object is an instance of a specified class. By using this operator, one can verify the type of an object at runtime, which proves useful for ensuring correct type handling and for enabling safe typecasting in a programme.

```
1  String name = "James";
2  System.out.println(name instanceof String); // true
```

Operator Precedence & Associativity

- Operator precedence determines which operator is evaluated first in an expression.
- Higher precedence operators (like * and /) are evaluated before lower precedence ones (like + and -).
- Associativity defines evaluation order when operators have the same precedence.

Example:

```
1  int result = 10 + 5 * 2; // result = 20 (because * has higher precedence)
2  int result2 = (10 + 5) * 2; // result2 = 30
```

Decision-Making

Understanding Decision-Making Statements

Decision-making statements form the backbone of a programme's control flow. These statements enable the programme to evaluate given conditions and, based on those evaluations, execute specific blocks of code. By doing so, they help determine the direction in which the programme proceeds,

allowing for dynamic choices during runtime. This capability is crucial for building logical operations and ensuring that the programme responds appropriately to different scenarios as they occur.

Types of Decision-Making Statements

if Statement

The **if statement** is used to execute a specific block of code only when a given condition evaluates to true. This enables the programme to make decisions and perform actions selectively, based on the result of the condition.

```
1  if (age >= 18) {
2      System.out.println("Eligible to vote.");
3  }
```

if...else Statement

The **if...else statement** enhances decision-making by providing an alternative block of code to execute when the condition is false. Thus, if the initial condition is not satisfied, the programme will automatically execute the alternative set of instructions, ensuring that only one of the blocks is run based on the condition.

```
1  if (age >= 18) {
2      System.out.println("Eligible to vote.");
3  } else {
4      System.out.println("Not eligible.");
5  }
```

Nested if Statement

Nested if statements allow for the evaluation of multiple conditions by placing one if or else if statement inside another. This structure enables the programme to check for a series of conditions in a specific sequence, ensuring that each condition is only evaluated if the previous conditions are met or not met. By using nested if statements, developers can construct more complex decision-making logic within their programmes, facilitating precise control over which blocks of code are executed under various scenarios.

```
1  if (score >= 90) {
2      System.out.println("Grade A");
3  } else if (score >= 75) {
4      System.out.println("Grade B");
5  } else {
6      System.out.println("Grade C");
7  }
```

switch Statement

A variable is compared with several possible values (cases), and the corresponding block is executed when a match is found.

```
int day = 3;
switch (day) {
    case 1: System.out.println("Monday"); break;
    case 2: System.out.println("Tuesday"); break;
    case 3: System.out.println("Wednesday"); break;
}
``
```

Java Loop Control Structures

Loops allow for the repeated execution of a block of code until a specified condition is met. They are commonly used for processing data, performing repetitive tasks, and automating sequences of operations within a programme.

Types of Loops

for Loop

The **for loop** is ideal when the number of iterations is known in advance. It consists of an initialisation step, a condition that is checked before each iteration, and an increment or decrement operation. The loop continues to execute the block of code as long as the condition remains true.

```
for (int i = 1; i <= 5; i++) {
    System.out.println(i);
}
```

for-each Loop

The **for-each loop** simplifies the process of iterating over arrays or collections. Instead of managing index variables, the loop automatically traverses each element, making the code easier to read and less error-prone.

```
int[] numbers = {1, 2, 3};
for (int num : numbers) {
    System.out.println(num);
}
```

while Loop

The **while loop** is used when the number of iterations is not known beforehand. It repeatedly executes a block of code as long as the given condition evaluates to true. The condition is checked before each execution, ensuring that the loop may not run even once if the condition is initially false.

```
int i = 1;
while (i <= 5) {
    System.out.println(i);
    i++;
}
```

do-while Loop

The **do-while loop** ensures that the block of code is executed at least once before the condition is evaluated. This is because the condition is checked after the execution of the loop body, guaranteeing a minimum of one iteration regardless of the initial state of the condition.

```java
int i = 1;
do {
    System.out.println(i);
    i++;
} while (i <= 5);
```

Loop Control Statements

continue Statement

The **continue statement** is used within loops to skip the current iteration and proceed directly to the next iteration of the loop. When the continue statement is encountered, the remaining code inside the loop body for that particular iteration is not executed, and the loop moves on to evaluate its next cycle.

```java
for (int i = 1; i <= 5; i++) {
    if (i == 3) continue;
    System.out.println(i);
}
```

break Statement

The **break statement** is used to immediately terminate the execution of a loop. When the break statement is encountered within the loop body, control is transferred outside the loop, and the loop ceases to execute further iterations.

```java
for (int i = 1; i <= 10; i++) {
    if (i == 5) break;
    System.out.println(i);
}
```

OOPs (Object-Oriented Programming)

Object-Oriented Programming (OOP) in Java is a programming paradigm that structures software design around **objects**. Unlike approaches that focus mainly on functions and logic, OOP emphasises entities that represent real-world objects, encapsulating both data and behaviour.

Key Principles of OOP

- **Data Encapsulation:** OOP ensures that data is bundled together with the methods that operate on that data, protecting it from outside interference and misuse.
- **Modularity:** Programmes are divided into distinct components or objects, each handling a specific functionality. This modular approach makes it easier to understand and manage complex systems.
- **Reusability:** OOP enables the creation of reusable code through inheritance, where new objects can be based on existing ones, reducing redundancy and improving efficiency.

By focusing on these core aspects, Object-Oriented Programming in Java enhances the maintainability and scalability of programmes, contributing to the development of robust and flexible software solutions.

Classes

A class in Java is a blueprint for creating objects, defining their data (attributes or fields) and actions (methods). By specifying these in a class, developers can make multiple objects with shared structure and behaviour.

WHY IT IS USED:

Classes provide structure and modularity in programming. They allow developers to group related data and operations together, making code reusable and easier to maintain.

REAL-TIME USAGE:

In an e-commerce application, a Product class can define properties like name, price, and methods like calculateDiscount(). Every product object will follow this structure.

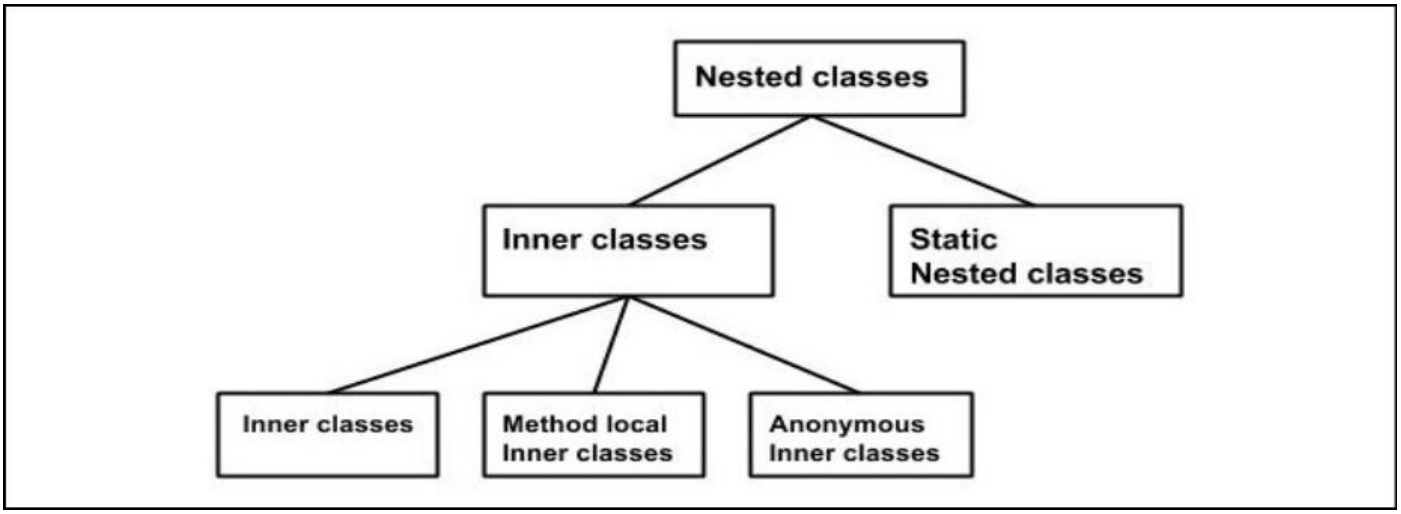
Properties of Java Classes

- A class does not consume memory resources until an instance (object) is instantiated.
- It serves as a blueprint rather than representing a tangible entity. Classes consist of methods and data members and may also contain nested classes.
- They facilitate all core object-oriented programming principles, including inheritance, encapsulation, and abstraction.

Types of Classes in Java

Nested Classes

An inner class is a class defined inside another class. It is used to logically group classes that are only used in one place, and it provides better encapsulation and readability. Inner classes can access the members of the outer class, even private ones.



JAVA SUPPORTS TWO TYPES OF NESTED CLASSES:

1. Non-Static Nested Classes (Inner Classes)

Definition:

A non-static nested class is called an inner class. It is associated with an instance of the outer class.

Characteristics:

- Can access instance variables and methods of the outer class.
- Requires an object of the outer class to create an instance.

2. Static Nested Classes

Definition:

A static nested class is declared with the static keyword. It does not require an instance of the outer class.

```
class Outer {
    static int data = 30;
    static class Inner {
        void show() {
            System.out.println("Data: " + data);
        }
    }
    public static void main(String[] args) {
        Outer.Inner inner = new Outer.Inner();
        inner.show();
    }
}
```

Characteristics:

- ❖ Can access static members of the outer class directly.
- ❖ Cannot access instance members without an object.

Types of Java Inner Classes

1. Member Inner Class

A member inner class is a non-static class defined inside another class. It is associated with an instance of the outer class and can access all members of the outer class, including private ones.

```
class Outer {
    private String message = "Hello from Outer";

    class Inner {
        void display() {
            System.out.println(message); // Accessing outer class private member
        }
    }

    public static void main(String[] args) {
        Outer outer = new Outer();
        Outer.Inner inner = outer.new Inner();
        inner.display();
    }
}
```

To logically group classes that are only used by the outer class and to improve encapsulation.

2. Local Inner Class

A local inner class is defined inside a method, constructor, or block. It can access local variables of the enclosing method only if they are declared final or effectively final.

It is used for temporary, method-specific functionality.

```java
class Outer {
    void display() {
        final String msg = "Local Inner Class Example";

        class Inner {
            void print() {
                System.out.println(msg);
            }
        }

        Inner inner = new Inner();
        inner.print();
    }

    Run | Debug
    public static void main(String[] args) {
        new Outer().display();
    }
}
```

3. Anonymous Inner Class

An anonymous inner class is a class without a name, created for one-time use. It is often used to implement interfaces or extend classes inline.

It is used for quick implementation without creating a separate class file, commonly used in event handling.

```java
interface Greeting {
    void sayHello();
}

public class Test {
    public static void main(String[] args) {
        Greeting g = new Greeting() {
            public void sayHello() {
                System.out.println("Hello from Anonymous Inner Class");
            }
        };
        g.sayHello();
    }
}
```

Singleton Class

A Singleton class ensures that only one instance of the class is created and provides a global point of access to that instance. This is achieved by making the constructor private, storing the instance in a static variable, and providing a public static method to access it. Singleton is a creational design pattern and is widely used in scenarios where a single shared resource is needed across the application.

Advantages of Singleton Design Pattern

Controlled Access to Instance

Ensures that only one object is created, reducing memory usage and avoiding redundant initialization.

Global Access Point

The instance can be accessed globally via a static method, making it easy to use across different classes.

Lazy Initialization

The object is created only when needed, improving performance and resource management.

Thread Safety

With proper synchronization or use of Enums, Singleton can be made thread safe.

Prevents Multiple Instantiations

Useful in managing shared resources like configuration settings, logging, or database connections.

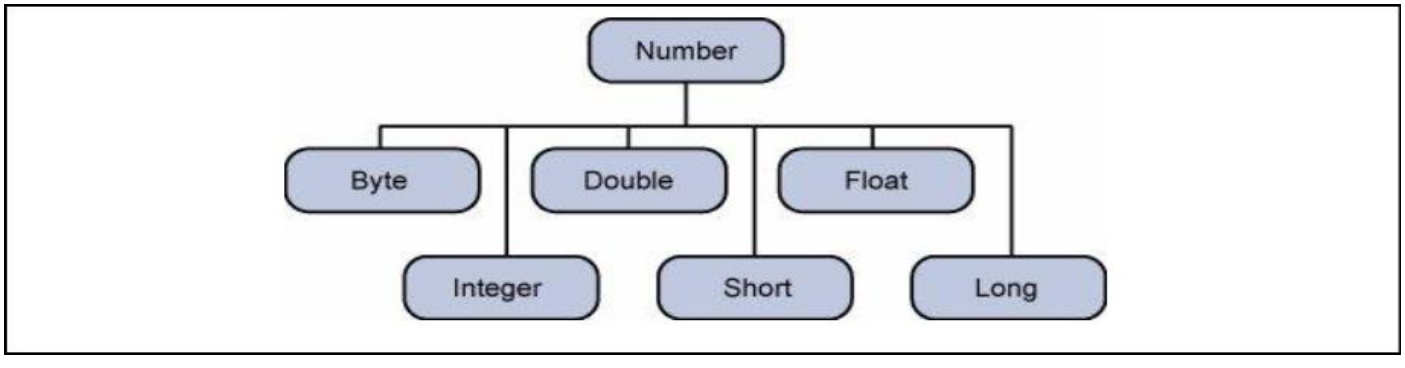
Use Cases of Singleton Pattern

Use Case	Description	Example
Logger	Centralized logging mechanism	Logger.getInstance().log("message");
Configuration Manager	Load and manage app settings	ConfigManager.getInstance().get("db.url");
Database Connection Pool	Manage shared DB connections	DBPool.getInstance().getConnection();
Cache Manager	Store and retrieve cached data	Cache.getInstance().put("key", value);
Thread Pool Manager	Manage threads efficiently	ThreadPool.getInstance().execute(task);

Wrapper Classes

Wrapper classes in Java are object representations of primitive data types. They allow primitives like int, char, boolean, etc., to be treated as objects. This is especially useful when working with collections, generics, or object-oriented features that require objects rather than primitives.

All the wrapper classes (Integer, Long, Byte, Double, Float, Short) are subclasses of the abstract class Number.



Object of Java Wrapper Class

The object of the wrapper class contains or wraps its **respective primitive data type**. Converting primitive data types into object is called **boxing**, and this is taken care by the compiler. Therefore, while using a wrapper class you just need to pass the value of the primitive data type to the constructor of the Wrapper class. And the Wrapper object will be converted back to a primitive data type, and this process is called unboxing. The Number class is part of the java.lang package.

AUTOBOXING

Autoboxing is the automatic conversion of a primitive type to its corresponding wrapper class.

UNBOXING

Unboxing is the automatic conversion of a wrapper class object back to its corresponding primitive type.

Notes

- Autoboxing and unboxing can lead to NullPointerException if a null wrapper object is unboxed.
- It may introduce performance overhead due to object creation and garbage collection.

List of Java Wrapper Classes

Primitive Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Types of Methods in Java

1. Instance Methods

Methods that belong to an object and can access instance variables directly. It is used to perform operations on object-specific data.

2. Static Methods

Methods declared with the static keyword. They belong to the class, not to any object. For utility operations or when no object-specific data is needed.

3. Abstract Methods

Methods declared without implementation in an abstract class or interface. It is used to enforce subclasses to provide specific implementations.

4. Final Methods

Methods declared with final keyword cannot be overridden by subclasses. It is used to prevent modification of critical behaviour.

5. Synchronized Methods

Methods declared with synchronized keyword to allow thread-safe operations. It is used in multi-threaded environments to prevent race conditions

6. Native Methods

Methods implemented in languages like C/C++ using JNI (Java Native Interface).It is used to interact with system-level resources or legacy code.

7. Concrete Methods

A concrete method is a method that has a complete implementation (method body) in a class.

- In regular classes (all methods are concrete unless declared abstract).
- In abstract classes, alongside abstract methods.

Concrete methods provide actual behaviour that can be reused or overridden by subclasses.

8. Default Methods (in Interfaces)

Introduced in Java 8, a default method in an interface has a method body. It is introduced to allow adding new methods to interfaces without breaking existing implementations.

- Declared using the default keyword.
- Can be overridden by implementing classes.

Interface

Java interface is a collection of abstract methods. The interface is used to achieve abstraction in which you can define methods without their implementations (without having the body of the methods). An interface is a reference type and is like the class.

Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods.

Writing an interface is like writing a class. However, a class describes the attributes and behaviours of an object. An interface contains behaviours that a class implements. Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

Java Interfaces and Classes: Similarities and Differences

Similarities

- Both define a blueprint for objects.
- Both can declare methods that describe behaviour.
- Both support inheritance (classes extend other classes; interfaces extend other interfaces).

Differences

Feature	Class	Interface
Implementation	Can have concrete methods and fields	Methods are abstract by default (Java 8+ allows default & static)
Variables	Instance variables allowed	Only constants (`public static final`)
Inheritance	Single inheritance	Multiple inheritance allowed
Constructors	Can have constructors	Cannot have constructors

Properties of Java Interface

- All fields are public static final by default.
- All methods are public and abstract by default (except default and static methods introduced in Java 8).
- Interfaces cannot be instantiated.
- Interfaces support multiple inheritance.
- Interfaces can have:
  - Abstract methods
  - Default methods (Java 8+)
  - Static methods (Java 8+)
  - Private methods (Java 9+ for internal use)

Tagging (Marker) Interfaces

A marker interface is an interface with no methods or fields.

Used to tag classes for special behaviour (e.g., Serializable, Cloneable).

Objects

An object is an instantiation of a class, serving as a representation of a real-world entity. It encapsulates both state (data) and behaviour (methods).

WHY IT IS USED:

Objects allow interaction with the data and methods defined in a class. They make programs dynamic and enable object-to-object communication.

REAL-TIME USAGE:

In a banking system, an Account object can store account details and perform operations like deposit() or withdraw().

Key Characteristics of Objects

State: The state of an object is stored in its fields, such as variables that hold data. For example, fields like *name* and *age* represent the specific information associated with an object.

Behaviour: The behaviour of an object is defined by its methods. These methods, such as *run()* or *bark()*, determine how the object acts or responds to messages.

Identity: Every object has its own unique reference in memory, which distinguishes it from all other objects, even if they have identical state and behaviour.

Creating Java Objects

Objects are created in Java using the **new** keyword. The process involves three main steps: **declaration**, **instantiation**, and **initialization**.

Declaration involves defining a variable of the class type, instantiation allocates memory using the **new** keyword, and initialization calls the constructor to set up the object. These steps are typically combined in a single line of code for simplicity and clarity. Example:

```
Person p = new Person();
```

combines declaration (`Person p`), instantiation (`new Person`), and initialization (calling the `Person` constructor).

Rules for Classes and Objects

Only one public class per source file.

This ensures clarity and prevents naming conflicts during compilation.

For example: `public class Student { }`.

File name must match the public class name.

This convention allows the compiler to easily locate the corresponding source file for a public class. For instance, if your public class is `Student`, the file should be named `Student.java`.

Package statement (if any) should be the first line.

This specifies the package to which the class belongs, helping to organise code and avoid naming conflicts. Example: `package com.example.project;`

Import statements come after package and before class declaration.

Imports allow you to use classes from other packages and must be placed before the class declaration to ensure dependencies are correctly resolved. Example: `import java.util.List;`

All classes in a file share the same package and imports.

This maintains consistency, making it easier to manage code dependencies and structure within a project. For example, if you declare multiple classes in a single file, they will all belong to the same package and can access the same imported classes.

Special Types of Classes

Abstract Classes**:** Cannot be instantiated; used for abstraction.

```
abstract class Animal {
    abstract void makeSound();
}
class Dog extends Animal {
    void makeSound() { System.out.println("Bark"); }
}
```

Final Classes**:** Cannot be extended.

```
final class MathUtils {
    static int add(int a, int b) { return a + b; }
}
```

Inner Classes: Defined inside another class.

```
class Outer {
    class Inner {
        void display() { System.out.println("Inner class method"); }
    }
}
```

Anonymous Classes: Created without a name for one-time use.

```
Runnable r = new Runnable() {
    public void run() {
        System.out.println("Running");
    }
};
```

Why Classes and Objects are Important

Classes and objects form the foundation of Object-Oriented Programming (OOP) in Java.

Classes allow developers to organise code into reusable modules, making it easier to maintain and scale applications as they grow. This approach also encourages modularity, as well as reusability and scalability in software development.

They help represent real-world entities in software design, bridging the gap between conceptual models and practical implementation.

Class Attributes

Java class attributes are variables declared inside a class that define the state of objects created from that class.

Attributes store object-specific data and allow methods to operate on this data. They represent the properties of an object.

In a Student Management System, attributes like rollNo, name, and age in a Student class represent each student's details.

Syntax:

```
access_modifier data_type attribute_name;
```

Example:

```
public class Dog {
    String breed;
    int age;
    String color;
}
```

Here, breed, age, and colour are class attributes.

USE THE FINAL KEYWORD TO MAKE ATTRIBUTES IMMUTABLE. THIS IS USEFUL FOR CONSTANTS LIKE MAX_SPEED IN A CAR CLASS.

Java Class Methods

Methods in a class specify the actions or behaviour that objects can perform. They are blocks of code designed to carry out specific tasks and often operate on the attributes (data) of the class.

Why They Are Used

Methods enhance code reusability and modularity and are essential for object-oriented design by associating actions with data. This makes it easier to manage and update code.

Real-time Usage

For instance, in a banking system, a deposit() method updates an account's balance, while a withdraw() method deducts money. These methods represent real-world banking actions within your software.

Creating (Declaring) Java Class Methods

To declare a method in Java, specify the access modifier, return type, method name, and parameters (if any).

```
access_modifier return_type method_name(parameters) {
    // method body
}
```

```
public class Dog {
    String breed;
    void bark() {
        System.out.println("Dog is barking");
    }
}
```

Accessing Java Class Methods

Methods are accessed using an object reference and the dot operator. An **object reference** is a variable that points to (stores the address of) an object created from a class. The **dot operator (.)** is used to access the methods or attributes of the object.

```
Dog d = new Dog();
d.bark();
```

Here, account is the object reference, and deposit is accessed using the dot operator.

Inside the class, methods can call other methods directly without using the object reference or dot operator.

this Keyword in Java Class Methods

What it is: this is a special keyword in Java that refers to the current object—the instance on which a method is being called.

Why it is used: It is mainly used to distinguish between instance variables and local variables when they share the same name, and to pass the current object as a parameter to other methods or constructors.

```
class Student {
    String name;
    Student(String name) {
        this.name = name; // refers to instance variable
    }
}
```

Public vs. Static Class Methods

- **Public Methods:** These belong to individual objects and require creating an object before they can be called. For example, account.withdraw(500); is an object-specific action.
- **Static Methods:** These belong to the class itself and can be called without creating any objects. They are typically used for utility functions. For example, Math.sqrt(25); calculates the square root of 25.

Real-time Usage

Public methods are used for object-specific actions, such as withdraw() in a banking application.

Static methods are used for tasks that do not depend on object state, such as Math.sqrt() for mathematical calculations.

Command-Line Arguments

Passed to main() as a String array:

Used in CLI tools to pass file paths or configuration options.

```
1  public static void main(String[] args) {
2      System.out.println("First argument: " + args[0]);
3  }
```

Variable Arguments (Var-Args)

Allows passing variable number of arguments of the same type.

Logging frameworks use var-args for flexible message formatting.

```
1  void printNames(String... names) {
2      for (String name : names) {
3          System.out.println(name);
4      }
5  }
```

The finalize() Method

finalize() is a method that may be called by the Java **Garbage Collector** before destroying an object. The **Garbage Collector** is a system process that automatically frees up memory by removing objects that are no longer in use.

Why it is used: The finalize() method is commonly used to perform cleanup operations, such as closing open files or releasing resources.

```
1  @Override
2  protected void finalize() throws Throwable {
3      if (file != null) {
4          file.close();
5      }
6      super.finalize();
7  }
8
```

Why you should avoid finalize()

- **Unreliable timing:** The garbage collector (GC) controls when (or if) finalize() runs, making resource release unpredictable.
- **Performance cost:** Objects with finalize() need extra GC work, which can slow down memory management and cause backlogs.
- **Risky semantics:** Objects may be resurrected during finalization, leading to hard-to-find bugs and complicated GC behaviour.
- **Deprecation:** Finalization (finalize) was deprecated in Java 9 (JEP 421) and may be removed in future JDKs; it can even be disabled at runtime. It's now legacy and considered unsafe.

Preferred alternatives:

- **try-with-resources (Auto Closeable):** Ensures resources are closed automatically at the end of a block, preventing leaks and reducing errors.
- **Cleaner (from java.lang.ref):** Allows flexible cleanup actions without interfering with GC, avoiding finalization problems.
- **Explicit close() patterns (e.g., in finally):** Give developers direct control over resource management, making cleanup predictable and reliable.

Scope of Java Variables

In Java, the **scope** of a variable refers to the region of a program where the variable can be accessed. Understanding variable scope helps you write safer, more maintainable code. Java variables fall into three main categories: instance variables, class (static) variables, and local variables.

1. Instance Variables

Instance variables are declared inside a class but outside any method, constructor, or block. They represent the properties or state of an object.

Scope: These variables are accessible to all methods, constructors, and blocks within the class. If declared *public*, they can also be accessed from other classes using an object reference.

Lifetime: An instance variable exists as long as the object exists. It is stored in heap memory.

Real-time usage: For example, in a Student class, attributes like name and rollNo are instance variables used across multiple methods.

```
class Student {
    String name; // instance variable
    void display() {
        System.out.println(name);
    }
}
```

2. Class (Static) Variables

Static variables are declared with the static keyword inside a class. They belong to the class itself, not to any one object.

Scope: Static variables are accessible by all methods in the class and can be accessed using the class name, without creating an object.

Lifetime: A static variable exists for the entire duration of the program and is stored in static memory.

Real-time usage: In a banking system, a static variable like interestRate can be shared across all account objects.

```
class Bank {
    static double interestRate = 5.0;
}
```

3. Local Variables

Local variables are declared inside a method, constructor, or block. They are used for temporary storage during method execution.

Scope: These variables are accessible only within the method or block where they are declared.

Lifetime: A local variable exists only while the method or block is executing. It is stored in stack memory.

Real-time usage: For example, a counter variable inside a loop is local to that loop.

```
1  void calculate() {
2      int sum = 0; // local variable
3      System.out.println(sum);
4  }
```

Important Points About Variable Scope

Default (Package-Private) Access:

If no access modifier is specified (also known as 'default' or 'package-private'), the variable is accessible only within the same package. This distinction is important: 'default' means no modifier is used and equates to package-private access.

Access Levels in Java:

- public → Accessible everywhere.
- private → Accessible only within the class.
- protected → Accessible within the package and subclasses.
- default/package-private → Accessible within the same package.

Interface Fields:

Always public, static, and final by default. This means that any variable declared in an interface is automatically a constant (cannot be changed), belongs to the interface itself (not to instances), and is accessible from anywhere.

Best Practice:

Use private instance variables with **getters** and **setters** for encapsulation.

Java Constructors

A constructor in Java is a special method used to initialize objects when they are created. Unlike regular methods, constructors have the same name as the class and do not have a return type. They are automatically called when an object is instantiated using the new keyword. Constructors help set initial values for object attributes and prepare the object for use.

Rules for Creating Java Constructors

- The constructor's name must match the class name exactly.
- Constructors cannot have a return type, not even void.
- They can have access modifiers like public, private, or protected.
- If no constructor is explicitly defined, Java provides a default constructor automatically.
- Constructors can be overloaded, meaning multiple constructors with different parameter lists can exist in the same class.

Types of Java Constructors

Java supports three main types of constructors:

1. Default Constructor

A default constructor is provided by Java when no constructor is defined in the class. It initializes objects with default values (e.g., 0 for numbers, null for objects).

```
class Student {
    String name;
    int age;
    // Default constructor provided by Java
}
```

When you create Student s = new Student();, the default constructor runs.

2. No-Args Constructor

A no-argument constructor is explicitly defined by the programmer and does not take any parameters. It is often used to set custom default values or perform initialization logic.

```
class Student {
    Student() {
        System.out.println("No-Args Constructor Called");
    }
}
```

3. Parameterized Constructor

A parameterized constructor accepts arguments to initialize object attributes with specific values. This is useful when you want to create objects with custom data at the time of creation.

```
class Student {
    String name;
    int age;
    Student(String n, int a) {
        name = n;
        age = a;
    }
}
```

Constructor Overloading

Constructor overloading means defining multiple constructors in the same class with different parameter lists. This provides flexibility in object creation by allowing different ways to initialize an object.

```
class Student {
    Student() {
        System.out.println("Default Constructor");
    }
    Student(String name) {
        System.out.println("Name: " + name);
    }
    Student(String name, int age) {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}
```

Constructors simplify object initialization and ensure that objects start in a valid state. They are widely used in real-world applications such as:

- Initializing database connections in a DAO class.
- Setting up UI components in GUI applications.
- Preparing configuration objects in frameworks like Spring.

Access Modifiers:

Java access modifiers define the scope and visibility of classes, methods, constructors, and variables. They help in controlling access to data and functionality, ensuring security, encapsulation, and proper design in object-oriented programming.

Types of Access Modifiers in Java

Java provides four main access levels:

1. Default (Package-Private) Access Modifier

When no access modifier is specified for a member in Java, it receives default, also known as package-private, access. This means the member is visible only within the same package, restricting its accessibility outside of the package.

PURPOSE OF DEFAULT ACCESS

Default access is particularly useful for grouping related classes within the same package. It allows these classes to interact freely with one another while keeping implementation details hidden from classes in other packages. This approach supports modular design and maintains a level of encapsulation within the package structure.

2. Private Access Modifier

Members declared as private in Java are accessible only within the same class. This means that variables, methods, or constructors marked as private cannot be accessed or modified from outside their defining class.

WHY USED

The private access modifier restricts access to sensitive data, enforcing encapsulation and protecting the class's internal state from unwanted changes.

REAL-TIME USAGE

For example, in a banking application, the account balance should be declared as a private member. This ensures that the balance can only be accessed or modified through designated getter and setter methods, maintaining both security and proper data handling within the application.

3.Protected Access Modifier

Protected members can be accessed by classes within the same package, as well as by subclasses—even if those subclasses are in different packages.

PURPOSE:

This approach enables controlled inheritance, ensuring that only related classes have access while keeping the details hidden from unrelated classes.

4. Public Access Modifier

Public members are accessible from any part of the application.

PURPOSE:

These are intended for methods or classes meant to be accessible everywhere, like those commonly used in API development.

Key Points

- Fields declared within interfaces are, by default, always public, static, and final.
- Any methods defined in interfaces are inherently public and abstract unless otherwise specified.

Recommended Practice

Choose the most restrictive access level you need; for instance, opt for private when declaring fields whenever possible.

Access Modifier	Within Class	Within Package	Outside Package by Subclass Only	Outside Package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Inheritance:

Inheritance is one of the core principles of Object-Oriented Programming (OOP) in Java. It allows one class (called the child or subclass) to acquire properties and behaviours of another class (called the parent or superclass).

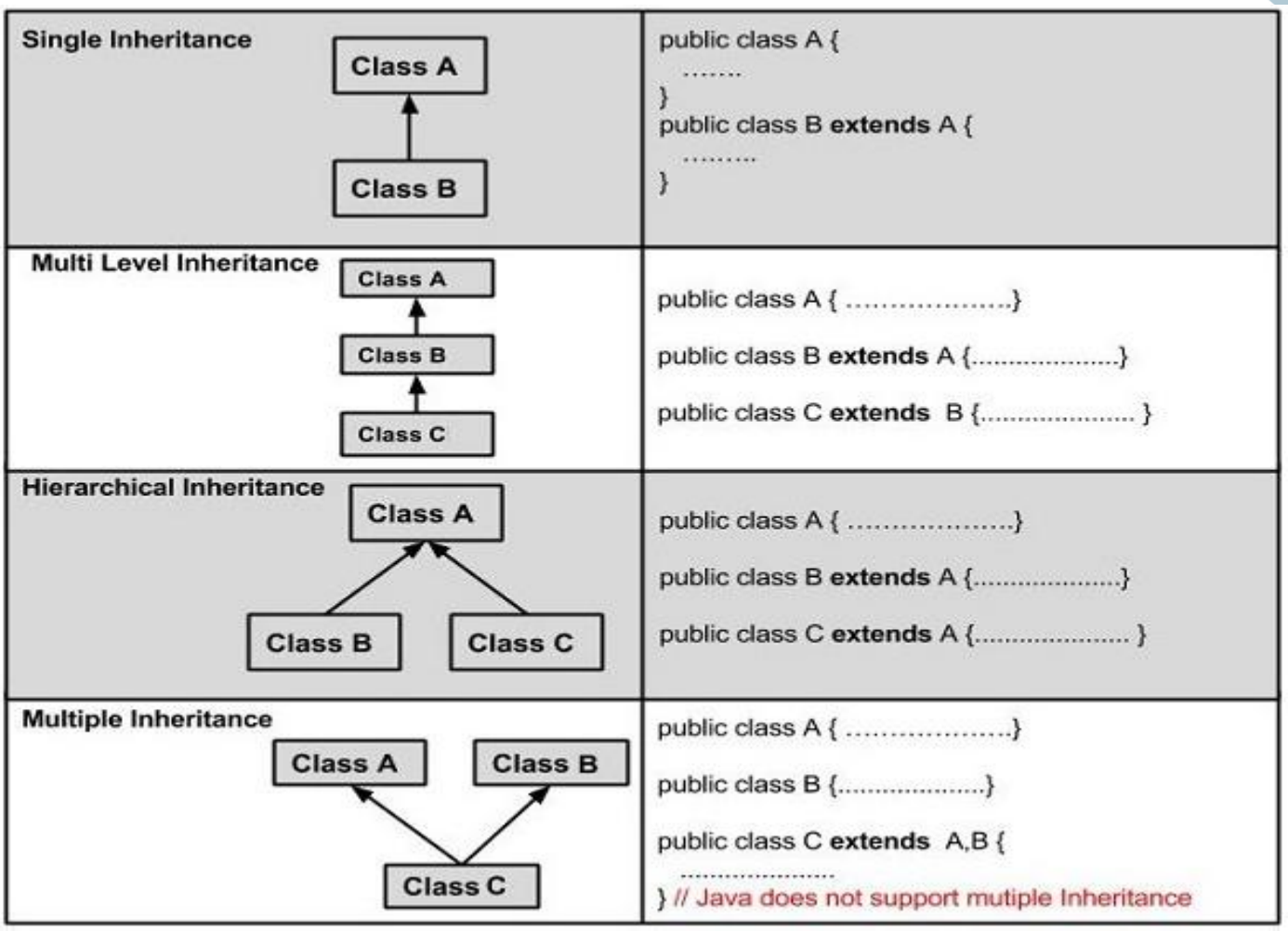
Need of Java Inheritance

- **Code reusability**: Inheritance allows you to reuse existing functionality in other classes or packages.
- **Extensibility**: It enables you to add or modify features in derived classes based on a base class.
- **Method overriding**: Inheritance is essential for implementing polymorphism through method overriding.
- **Abstraction**: Achieving abstraction in object-oriented programming requires inheritance.

KEY RULES OF INHERITANCE

- Use the **extends keyword** for class inheritance.
- A class can **inherit only one parent class** (Java does not support multiple inheritance for classes).
- All classes implicitly inherit from **Object class**.
- Constructors are **not inherited**, but the child class calls the parent constructor using super().
- Private members are **inherited but not accessible** directly.

Types of Inheritance in Java



1. Single Inheritance

The inheritance in which there is only one base class and one derived class is known as single inheritance. The single (or, single level) inheritance inherits data from only one base class to only one derived class.

```
1  class Vehicle {
2      void start() { System.out.println("Starting..."); }
3  }
4  class Car extends Vehicle {
5      void honk() { System.out.println("Honking..."); }
6  }
```

2. Multilevel Inheritance

The inheritance in which a base class is inherited to a derived class and that derived class is further inherited to another derived class is known as multi-level inheritance. Multilevel inheritance involves multiple base classes.

```
1  class Animal { void eat() { System.out.println("Eating"); } }
2  class Mammal extends Animal { void walk() { System.out.println("Walking"); } }
3  class Dog extends Mammal { void bark() { System.out.println("Barking"); } }
```

3. Hierarchical Inheritance

The inheritance in which only one base class and multiple derived classes is known as hierarchical inheritance.

```
class Shape { void draw() { System.out.println("Drawing"); } }
class Circle extends Shape {}
class Square extends Shape {}
```

4. Hybrid Inheritance (via Interfaces)

Java does not support multiple inheritance for classes but allows it through interfaces.

```
interface Printable { void print(); }
interface Showable { void show(); }
class Demo implements Printable, Showable {
    public void print() { System.out.println("Printing"); }
    public void show() { System.out.println("Showing"); }
}
```

IS-A Relationship VS HAS-A Relationship

IS-A Relationship (Inheritance)

The IS-A relationship represents inheritance in Java. It means that one class is a type of another class. This relationship is established using the extends keyword for classes or implements for interfaces. When a class inherits from another, it automatically gains access to the parent class's properties and methods, promoting code reusability and supporting polymorphism.

Consider a Car class that extends a Vehicle class. A car is a type of vehicle, so the IS-A relationship holds true. This allows the Car class to reuse the start() method from Vehicle without rewriting it.

```
class Vehicle {
    void start() { System.out.println("Vehicle starting..."); }
}
class Car extends Vehicle {
    void honk() { System.out.println("Car honking..."); }
}
```

Here, Car IS-A Vehicle. This relationship is ideal when you want to create a hierarchy of classes that share common behaviour.

HAS-A Relationship (Composition/Aggregation)

The HAS-A relationship represents composition or aggregation in Java. It means that one class contains another class as a field. Instead of inheriting behaviour, the class uses another class to perform certain tasks. This approach is often preferred over inheritance because it provides flexibility and avoids tight coupling.

Consider a Car class can have an Engine object. A car is not an engine, but it has an engine. This relationship allows the Car class to delegate engine-related tasks to the Engine class.

```java
class Engine {
    void startEngine() { System.out.println("Engine started"); }
}
class Car {
    Engine engine = new Engine(); // Car HAS-A Engine
    void startCar() {
        engine.startEngine();
        System.out.println("Car started");
    }
}
```

Here, Car HAS-A Engine. This design is widely used in real-world applications like dependency injection frameworks (Spring), where objects are composed rather than inherited.

KEY DIFFERENCES

IS-A (Inheritance):

Used when one class is a specialized version of another. It supports polymorphism and method overriding.

HAS-A (Composition):

Used when one class needs the functionality of another without being its subtype. It promotes loose coupling and better maintainability.

Polymorphism

Polymorphism is one of the fundamental principles of Object-Oriented Programming (OOP). The term means “many forms”, and in Java, it allows a single interface or method to represent different behaviours depending on the object that invokes it. This makes code flexible, reusable, and easier to maintain.

For example, consider a draw() method in a Shape class. Different subclasses like Circle and Rectangle can provide their own implementation of draw(). When you call draw() on a Shape reference, the actual method executed depends on the object type at runtime.

Use of Polymorphism in Java

Code Reusability: Write generic code that works for multiple object types.

Flexibility: Add new classes without changing existing code.

Frameworks: Polymorphism is heavily used in Java frameworks like Spring and Hibernate for dependency injection and dynamic behaviour.

Real-world example:

In a payment system, a processPayment() method can handle Credit Card, UPI, or Net Banking payments differently using polymorphism.

Types of Polymorphism in Java

1. Compile-Time Polymorphism (Static Binding)

The method to execute is determined at compile time. This is achieved through method overloading.

HOW IT WORKS:

Multiple methods in the same class share the same name but differ in parameter type, number, or order.

```java
class Calculator {
    int add(int a, int b) { return a + b; }
    double add(double a, double b) { return a + b; }
}
```

WHY IT MATTERS:

Improves readability and code reuse by allowing multiple versions of a method for different data types.

REAL-WORLD USAGE:

In printing utilities, print() can accept String, int, or double.

2. Run-Time Polymorphism (Dynamic Binding)

The method to execute is determined at runtime. This is achieved through method overriding.

HOW IT WORKS:

A subclass provides a specific implementation of a method already defined in its superclass.

```java
class Animal {
    void sound() { System.out.println("Animal sound"); }
}
class Dog extends Animal {
    void sound() { System.out.println("Bark"); }
}
```

```java
Animal a = new Dog();
a.sound(); // Output: Bark
```

WHY IT MATTERS:

Enables polymorphic behaviour and **dynamic method dispatch**, which is essential for frameworks and APIs.

REAL-WORLD USAGE:

In payment gateways, processPayment() behaves differently for CreditCard vs UPI.

DYNAMIC METHOD DISPATCH IS THE MECHANISM BY WHICH A CALL TO AN OVERRIDDEN METHOD IS RESOLVED AT RUNTIME, NOT AT COMPILE TIME.

WHEN A PARENT CLASS REFERENCE POINTS TO A CHILD CLASS OBJECT, THE METHOD THAT GETS EXECUTED DEPENDS ON THE ACTUAL OBJECT TYPE, NOT THE REFERENCE TYPE.

THIS IS POSSIBLE BECAUSE NON-STATIC METHODS IN JAVA ARE VIRTUAL BY DEFAULT.

Initializer Block

An initializer block is a block of code inside a class that runs when an object is created. It is used to initialize instance variables and perform setup tasks.

Java provides two types of initializer blocks:

- Instance Initializer Block (IIB) – Runs every time an object is created.
- Static Initializer Block – Runs only once when the class is loaded.

Static Initializer Block

A static initializer block is a block of code inside a class that runs only once when the class is loaded into memory. It is used to initialize static variables or perform class-level setup

Syntax:

```
class Config {
    static String appName;
    static {
        appName = "MyApp";
        System.out.println("Static block initialized appName");
    }
}
```

Characteristics:

- Executes before any object is created and before the main method runs.
- Runs only once per class loading.
- Cannot access instance variables directly (only static members).

Use Case:

Useful for initializing static resources, like database connections or configuration settings.

Instance Initializer Block (IIB)

An instance initializer block is a block of code enclosed in { } inside a class, **without any method or constructor declaration** that runs every time an object is created, before the constructor executes.

Syntax:

```
class Student {
    String name;
    {
        System.out.println("Common initialization logic");
    }
    Student() { System.out.println("Default Constructor"); }
    Student(String name) { this.name = name; System.out.println("Parameterized
Constructor"); }
}
```

Characteristics:

- Executes after instance variables are initialized but before the constructor.
- Runs every time an object is created.
- Can access instance variables directly.

Use Case:

Useful for common initialization logic shared by multiple constructors.

Order of Execution

1. Static Block → Runs once when the class is loaded.
2. Instance Initializer Block → Runs every time an object is created.
3. Constructor → Runs after the instance initializer block.

Abstraction

Abstraction is one of the core principles of Object-Oriented Programming (OOP). It refers to hiding the implementation details and showing only the essential features of an object. In Java, abstraction is achieved using abstract classes and interfaces.

Why it is used:

- To reduce complexity by exposing only what is necessary.
- To improve maintainability and flexibility.
- To enforce a contract for subclasses or implementing classes.

Real-world analogy:

When you drive a car, you use the steering wheel and pedals without knowing the internal engine mechanics. This is abstraction.

How to Achieve Abstraction in Java

Abstract Classes

An abstract class in Java is a class that cannot be **instantiated** directly. It is used to provide a **base** structure for subclasses and can contain both **abstract methods** (without implementation) and **concrete methods** (with implementation). Abstract classes allow you to define common behaviour that all subclasses share while leaving some methods for subclasses to implement.

WHY USE ABSTRACT CLASSES

When you want to share common behaviour among subclasses but still enforce some methods to be implemented by them.

Example: A Shape class can define a display() method and leave draw() as abstract for subclasses like Circle and Rectangle.

```java
public abstract class Shape {
    // Instance variables (state)
    private final String color;
    private final boolean filled;

    // Constructor (used to initialize instance state)
    protected Shape(String color, boolean filled) {
        this.color = color;
        this.filled = filled;
    }

    // Abstract methods (no implementation: must be provided by subclasses)
    public abstract double area();
    public abstract double perimeter();

    // Concrete method (has implementation; shared by all subclasses)
    public void describe() {
        System.out.printf(format: "Shape[color=%s, filled=%s]%n", color, filled);
    }

    // Concrete getters (also demonstrate encapsulation)
    public String getColor() {
        return color;
    }

    public boolean isFilled() {
        return filled;
    }
}
```

Interfaces

An interface in Java is **a fully abstract type** that defines a contract for classes to implement. Interfaces cannot have **instance variables or constructors**. All fields **are public static final**, and all methods are **public and abstract** by default (except default and static methods introduced in Java 8).

WHY USE INTERFACES

When you want to define capabilities that can be applied across unrelated classes.

Supports multiple inheritance, which abstract classes do not.

Example: A Drawable interface can be implemented by both Circle and Rectangle classes.

```java
// File: Movable.java
public interface Movable {
    // Abstract methods (no implementation)
    void moveBy(double dx, double dy);
    double[] position(); // returns [x, y]

    // Default method (has implementation; can be overridden by implementers)
    default void moveTo(double x, double y) {
        double[] pos = position();
        moveBy(x - pos[0], y - pos[1]);
    }

    // Static utility method (belongs to interface type, not instances)
    static double distance(Movable a, Movable b) {
        double[] pa = a.position();
        double[] pb = b.position();
        double dx = pa[0] - pb[0];
        double dy = pa[1] - pb[1];
        return Math.hypot(dx, dy);
    }
}
```

Encapsulation

Encapsulation is one of the four pillars of Object-Oriented Programming (OOP). It refers to binding data (variables) and methods that operate on that data into a single unit (class) and restricting direct access to the data.

This is achieved by:

- Declaring fields as private.
- Providing public getter and setter methods for controlled access.

WHY ENCAPSULATION?

- Data Security: Prevents unauthorized access to sensitive data.
- Flexibility: Allows validation before updating fields.
- Maintainability: Internal implementation can change without affecting external code.

Fully Encapsulated Class

A class in which all data members are declared as private, granting access exclusively via getter and setter methods.

Partially Encapsulated Class

A class in which certain fields are designated as private, while others may be public or protected.

EXCEPTION HANDLING

Exception: An unwanted unexpected event that disturbs normal flow of the program is called exception.

Example:

- SleepingException
- TyrePunchuredException
- FileNotFoundException ...etc

It is highly recommended to handle exceptions. The main objective of exception handling is graceful (normal) termination of the program.

MEANING OF EXCEPTION HANDLING

Exception handling doesn't mean repairing an exception. We have to define alternative way to continue rest of the program normally. This way of defining alternative is nothing but exception handling.

1.FOR EVERY THREAD JVM WILL CREATE A SEPARATE STACK AT THE TIME OF THREAD CREATION. ALL METHOD CALLS PERFORMED BY THAT THREAD WILL BE STORED IN THAT STACK. EACH ENTRY IN THE STACK IS CALLED "ACTIVATION RECORD" (OR) "STACK FRAME".

2.AFTER COMPLETING EVERY METHOD CALL JVM REMOVES THE CORRESPONDING ENTRY FROM THE STACK.

3.AFTER COMPLETING ALL METHOD CALLS JVM DESTROYS THE EMPTY STACK AND TERMINATES THE PROGRAM NORMALLY.

Default Exception Handling in Java

Java provides a built-in mechanism to handle exceptions when the programmer does not explicitly handle them using try-catch. This process, called default exception handling, is overseen by the JVM (Java Virtual Machine).

Step-by-Step Breakdown

1.EXCEPTION OBJECT CREATION

- When an exception occurs inside a method, that method creates an Exception object.
- This object contains:
  - Name of the exception (e.g., ArithmeticException)
  - Description/message (e.g., / by zero)
  - Location (stack trace showing where the exception occurred)

2.HANDING OVER TO JVM

- The method hands over the exception object to the JVM.

3.JVM CHECKS FOR HANDLING CODE

- JVM checks if the method has any try-catch block to handle the exception.
- If not, the method is terminated abnormally, and its stack frame is removed.

4.CALLER METHOD CHECK

- JVM then checks the caller method for exception handling code.
- If the caller also lacks handling code, it is terminated, and its stack frame is removed.

5.PROPAGATION TO MAIN()

- ❖ This process continues up the call stack until it reaches the main() method.
- ❖ If main() also lacks handling code, it is terminated.

6.DEFAULT EXCEPTION HANDLER

- ❖ JVM invokes the default exception handler.
- ❖ This handler prints the exception details to the console in the following format:

```java
public class StackDemo {
    Run | Debug
    public static void main(String[] args) {
        int x = 10 / 0; // No try-catch
    }
}
```

- ❖ Exception:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at StackDemo.main(StackDemo.java:3)
```

Exception Hierarchy

At the top of the hierarchy is Throwable, which is the superclass for all errors and exceptions in Java.

Throwable has two direct subclasses:

- ❖ Error: Represents serious problems that applications should not try to handle (e.g., OutOfMemoryError, StackOverflowError).
- ❖ Exception: Represents conditions that applications might want to catch and handle.

Hierarchy Overview:

```
java.lang.Object
   └── java.lang.Throwable
        ├── java.lang.Error
        │    └── (Various Error subclasses)
        └── java.lang.Exception
             ├── java.lang.RuntimeException
             │    └── (Various unchecked exceptions)
             └── (Checked exceptions like IOException, SQLException)
```

Error vs Exception

Errors and exceptions serve different purposes in Java.

1.**Errors** indicate severe problems that typically cannot be recovered from, such as hardware failures or JVM crashes. For example, if the JVM runs out of memory, it throws an OutOfMemoryError, and the application cannot do much to recover.

2.**Exceptions**, however, represent abnormal conditions that occur during program execution but can often be anticipated and handled. For instance, trying to read a file that does not exist will throw an IOException, which can be caught and handled by notifying the user or retrying the operation. In short, errors are system-level issues, while exceptions are application-level issues.

Checked vs Unchecked Exceptions

- **Checked** exceptions are those that the compiler forces you to handle explicitly, either by using a **try-catch** block or by declaring them in the method signature with the throws keyword.
- These exceptions usually represent recoverable conditions, such as missing files or database connectivity issues.
- Examples include IOException and SQLException.
- **Unchecked** exceptions, on the other hand, do not require explicit handling. They include all subclasses of RuntimeException (such as NullPointerException, ArithmeticException) and all subclasses of Error.
- These exceptions typically represent programming mistakes or system failures.
- The compiler only enforces handling for checked exceptions.

Note:

- RuntimeException and its child classes, Error and its child classes are unchecked and all the remaining are considered as checked exceptions.
- Whether exception is checked or unchecked compulsory it should occurs at runtime only and there is no chance of occurring any exception at compile time.

Fully checked Vs Partially checked

A checked exception is said to be fully checked if and only if all its child classes are also checked.

Example:

- IOException
- InterruptedException

A checked exception is said to be partially checked if and only if some of its child classes are unchecked.

Example:

- Exception

Customized Exception Handling

try-catch block

Customized exception handling means writing your own logic to deal with errors in a way that makes sense for your application. Instead of letting the program terminate abruptly when an error occurs, you use **try-catch** blocks to catch exceptions and respond appropriately. This approach improves user experience and makes your code more robust.

The **try** block contains the code that **might throw an exception**. If an exception occurs, control **immediately jumps** to the corresponding **catch** block. Inside the catch block, you can write custom messages, log the error, or even recover from the problem.

```
try {
    // Code that might throw an exception
} catch (ExceptionType e) {
    // Customized handling logic
}
```

Various methods to print exception information

1. printStackTrace()

Description: Prints the complete stack trace of the exception, including the exception type, message, and the sequence of method calls that led to the error.

Usage: Best for debugging because it shows where the exception occurred.

```
try {
    int a = 10 / 0;
} catch (ArithmeticException e) {
    e.printStackTrace();
}
```

Output

```
java.lang.ArithmeticException: / by zero
    at CustomExceptionHandling.main(CustomExceptionHandling.java:3)
```

2. getMessage()

Description: Returns only the description message of the exception.

Usage: Useful when you want a short message without stack trace.

```
try {
    int[] arr = new int[2];
    System.out.println(arr[5]);
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Message: " + e.getMessage());
}
```

Output:

```
Message: Index 5 out of bounds for length 2
```

3. toString()

Description: Returns a string representation of the exception, which includes the exception class name and message.

Usage: Good for logging concise error info.

```
try {
    String str = null;
    System.out.println(str.length());
} catch (NullPointerException e) {
    System.out.println("Exception Info: " + e.toString());
}
```

Output:

```
Exception Info: java.lang.NullPointerException
```

4. getLocalizedMessage()

Description: Returns a localized description of the exception message (useful for internationalization).

```java
try {
    throw new Exception("Custom error occurred");
} catch (Exception e) {
    System.out.println("Localized Message: " + e.getLocalizedMessage());
}
```

Output

```
Localized Message: Custom error occurred
```

Finally block

- It is not recommended to take clean up code inside try block because there is no guarantee for the execution of every statement inside a try.
- It is not recommended to place clean up code inside catch block because if there is no exception then catch block won't be executed.
- We require some place to maintain clean up code which should be executed always irrespective of whether exception raised or not raised and whether handled or not handled. Such type of best place is nothing but finally block.
- Hence the main objective of finally block is to maintain cleanup code.

return Vs finally:

- Even though return statement present in try or catch blocks first finally will be executed and after that only return statement will be considered. i.e. finally block dominates return statement.
- If return statement present try, catch and finally blocks then finally block return statement will be considered.

finally vs System.exit(0):

- There is only one situation where the finally block won't be executed is whenever we are using System.exit(0) method. Whenever we are using System.exit(0) then JVM itself will be shutdown , in this case finally block won't be executed.
- i.e., System.exit(0) dominates finally block.

System.exit(0)

1. This argument acts as status code. Instead of zero, we can take any integer value
2. zero means normal termination , non-zero means abnormal termination
3. This status code internally used by JVM, whether it is zero or non-zero there is no change in the result and effect is same wrt program.

Notes

1.If we are not entering into the try block then the finally block won't be executed. Once we entered the try block without executing finally block we can't come out.

2. We can take try-catch inside try i.e., nested try-catch is possible

3. The most specific exceptions can be handled by using inner try-catch and generalized exceptions can be handle by using outer try-catch.

throw statement

Sometimes we can create Exception object explicitly and we can hand over to the JVM manually by using throw keyword.

```
throw new ExceptionType("Error message");
```

- **throw**: To handover our created exception object to jvm manually
- **new ExceptionType():** Creates an instance of an exception class (e.g., IOException, ArithmeticException).
- **Error message:** Optional message describing the exception.

Examples:

```
class Test {
    public static void main(String[] args) {
        System.out.println(10/0);
    }
}
```

- **Behaviour**: Division by zero occurs.
- **Exception** Handling: The JVM automatically creates an ArithmeticException object and throws it.
- **Key Point**: Exception creation and handover to JVM are automatic.

```
class Test {
    public static void main(String[] args) {
        throw new ArithmeticException("/ by zero");
    }
}
```

- **Behaviour**: Explicitly throws an ArithmeticException.
- **Exception Handling**: The programmer manually creates the exception object and throws it using throw.
- **Key Point**: Exception creation and handover to JVM are manual.

IN GENERAL WE CAN USE THROW KEYWORD FOR CUSTOMIZED EXCEPTIONS BUT NOT FOR PREDEFINED EXCEPTIONS.

Case 1 — throw e; when e is null

Rule: If you execute throw e; and e is null, Java throws a **NullPointerException** at **runtime**.

Why? The throw statement expects a reference to a **Throwable**. If that reference is null, the JVM throws NullPointerException immediately when throw executes (before control could transfer anywhere else).This is a **runtime** behaviour, not a compile-time error.

```
class Demo1 {
    public static void main(String[] args) {
        RuntimeException e = null;   // e refers to null
        // Some logic...
        throw e;                     // <-- runtime: NullPointerException
    }

```

Case 2 — No statements are allowed after a throw in the same block

Rule: Any statement that is guaranteed unreachable after a throw causes a compile-time error: **"unreachable statement".**

```
class Demo2 {
    public static void main(String[] args) {
        if (true) {
            throw new IllegalArgumentException("bad");
            // System.out.println("Hi"); // <-- compile-time error: unreachable statement
        }
    }
}

```

Case 3 — throw works only with Throwable types

Rule: The expression after throw must be of type Throwable (or a subclass). Otherwise, you get a compile-time error: "incompatible types: ... cannot be converted to Throwable".

```
class Demo3 {
    public static void main(String[] args) {
        Object o = new Object();
        // throw o; // <-- compile-time error: incompatible types: Object cannot be converted to Throwable
    }
}

```

Valid example (custom exception):

```
class MyAppException extends Exception {
    MyAppException(String msg) { super(msg); }
}
class Demo3Ok {
    static void work() throws MyAppException {
        throw new MyAppException("failed");
    }
}
```

Throws statement:

In our program if there is any chance of raising checked exception then compulsory we should handle either by try catch or by throws keyword otherwise the code won't compile.

```
returnType methodName(parameters) throws ExceptionType1, ExceptionType2 {
    // method body
}
```

Example:

```
class Test3 {
    public static void main(String[] args) throws InterruptedException {
        Thread.sleep(5000);       // exception is not handled here
    }
}
// Output: Compiles and runs (but main may terminate if actually interrupted)
```

You **declare** the method **may throw** InterruptedException. Responsibility is **delegated** to the caller (here, the JVM—the program will terminate with a stack trace if an interrupt occurs and is not caught higher up).

Key Points

PURPOSE:

It does not throw the exception itself; it only declares that the method may throw certain exceptions.

Actual throwing is done using the throw statement inside the method.

CHECKED EXCEPTIONS:

Required for checked exceptions (e.g., IOException, InterruptedException).

Unchecked exceptions (RuntimeException and subclasses) can be declared but are not required.

MULTIPLE EXCEPTIONS:

You can declare multiple exceptions separated by commas

EFFECT:

If a method declares throws InterruptedException, the caller must either:

- Handle it with try–catch
- Declare it again with throws.

Exception handling keywords summary:

- **try**: To maintain risky code.
- **catch**: To maintain handling code.
- **finally**: To maintain cleanup code.
- **throw**: To handover our created exception object to the JVM manually.
- **throws**: To delegate responsibility of exception handling to the caller method.

Various possible compile time errors in exception handling:

- Exception XXX has already been caught.
- Unreported exception XXX must be caught or declared to be thrown.
- Exception XXX is never thrown in body of corresponding try statement.
- Try without catch or finally.
- Catch without try.
- Finally without try.
- Incompatible types.
- found:Test
- requried:java.lang.Throwable;
- Unreachable statement.

Customized Exceptions (User defined Exceptions)

Sometimes we can create our own exception to meet our programming requirements. Such type of exceptions are called customized exceptions (user defined exceptions).

Example:

InSufficientFundsException, TooYoungException, TooOldException

Checked vs Unchecked: Which should you extend

CHECKED EXCEPTIONS → extend Exception (not RuntimeException).

Must be caught or declared (throws). Use for recoverable situations where callers are expected to handle the problem.

```java
// Step 1: Define custom exception
class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
}
```

```java
public class CustomExceptionDemo {
    public static void main(String[] args) {
        try {
            validateAge(15);
        } catch (InvalidAgeException e) {
            System.out.println("Caught custom exception: " + e.getMessage());
        }
    }

    static void validateAge(int age) throws InvalidAgeException {
        if (age < 18) {
            throw new InvalidAgeException("Age must be 18 or above");
        }
        System.out.println("Valid age");
    }
}
```

Output:

```
Caught custom exception: Age must be 18 or above
```

UNCHECKED EXCEPTIONS → extend RuntimeException.

Don't require throws. Use for programming errors or violations of invariants (e.g., illegal state, precondition failures).

```java
class NegativeBalanceException extends RuntimeException {
    public NegativeBalanceException(String message) {
        super(message);
    }
}
```

```java
public class BankAccount {
    private double balance;

    public void withdraw(double amount) {
        if (amount > balance) {
            throw new NegativeBalanceException("Insufficient funds!");
        }
        balance -= amount;
    }
}
```

Note: It is highly recommended to maintain our customized exceptions as unchecked by extending RuntimeException. We can catch any Throwable type including Errors also.

try with resources

Until 1.6 version it is highly recommended to write finally block to close all resources which are open as part of try block.

```
1  BufferedReader br = null;
2  try {
3      br = new BufferedReader(new FileReader("abc.txt"));
4      // use br based on our requirements
5  } catch (IOException e) {
6      // handling code
7  } finally {
8      if (br != null)
9          br.close();
10 }
```

The main advantage of "try with resources" is

the resources which are opened as part of try block will be closed automatically Once the control reaches end of the try block either normally or abnormally and hence we are not required to close explicitly so that the complexity of programming will be reduced. It is not required to write finally block explicitly and hence length of the code will be reduced and readability will be improved.

```
1  try (BufferedReader br = new BufferedReader(new FileReader("abc.txt"))) {
2      // use br based on our requirements
3  } catch (IOException e) {
4      // handling code
5  }
```

Multi catch block :

Until 1.6 version ,Even though Multiple Exceptions having same handling code we must write a separate catch block for every exception, it increases length of the code and reviews readability.

```
1  try {
2      // risky code
3  } catch (IOException e) {
4      e.printStackTrace();
5  } catch (SQLException e) {
6      e.printStackTrace();
7  }
```

To overcome this problem Sun People introduced "Multi catch block" concept in 1.7 version. The main advantage of multi catch block is we can write a single catch block , which can handle multiple different exceptions .

```
1  try {
2      // risky code
3  } catch (IOException | SQLException e) {
4      e.printStackTrace();
5  }
6
```

Exceptions in a multi-catch must not have a parent-child relationship.

Example: IOException | FileNotFoundException ❌ (invalid because FileNotFoundException is a subclass of IOException).

```java
public class ExceptionPropagationDemo {
    Run | Debug
    public static void main(String[] args) {
        try {
            method1();
        } catch (Exception e) {
            System.out.println("Exception handled in main: " + e.getMessage());
        }
    }

    static void method1() throws Exception {
        method2();
    }

    static void method2() throws Exception {
        method3();
    }

    static void method3() throws Exception {
        throw new Exception(message: "Error occurred in method3");
    }
}
```

Output:

```
Exception handled in main: Error occurred in method3
```

Rethrowing an Exception :

Rethrowing an Exception in Java means catching an exception in a catch block and then throwing it again (either the same exception or a new one) to be handled by another method higher up the call stack. This is useful when you want to log or perform some action locally but still let the caller know about the error.

```java
import java.io.IOException;
public class RethrowExample {
    Run | Debug
    public static void main(String[] args) {
        System.out.println(x: "\n");
        try {
            method1();
        } catch (IOException e) {
            System.out.println("Handled in main: " + e.getMessage());
        }
        System.out.println(x: "\n");
    }
    static void method1() throws IOException {
        try {
            method2();
        } catch (IOException e) {
            System.out.println(x: "Logging in method1");
            throw e; // rethrowing
        }
    }
    static void method2() throws IOException {
        throw new IOException(message: "File not found");
    }
}
```

Output:

```
Logging in method1
Handled in main: File not found
```

Multi-Threading

Multitasking

Executing several tasks simultaneously is the concept of multitasking. There are two types of multitasking's.

1. Process based multitasking.
2. Thread based multitasking.

Process based multitasking

Executing several tasks simultaneously where each task is a separate independent process such type of multitasking is called process based multitasking.

Example:

- While typing a java program in the editor we can be able to listen mp3 audio songs at the same time we can download a file from the net all these tasks are independent of each other and executing simultaneously and hence it is Process based multitasking.
- This type of multitasking is best suitable at "OS level"

Thread based multitasking

Executing several tasks simultaneously where each task is a separate independent part of the same program, is called Thread based multitasking. And each independent part is called a "Thread".

- This type of multitasking is best suitable for "programmatic level".
- When compared with "C++", developing multithreading examples is very easy in java because java provides in built support for multithreading through a rich API (Thread, Runnable, ThreadGroup, ThreadLocal...etc).
- In multithreading on 10% of the work the programmer is required to do and 90% of the work will be down by java API.
- Whether it is process based or Thread based the main objective of multitasking is to improve performance of the system by reducing response time.
- The main important application areas of multithreading are:
  - To implement multimedia graphics.
  - To develop animations.
  - To develop video games etc.
  - To develop web and application servers

The ways to define instantiate and start a new Thread

1. By extending Thread class.

2. By implementing Runnable interface.

DEFINING A THREAD BY EXTENDING "THREAD CLASS"

Create a class that extends Thread.

Override the run() method to define the task the thread will execute.

Instantiate the class, then call start() on the instance (not run() directly).

```java
// Defining a Thread by extending Thread class
class MyThread extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("child Thread");
        }
    }
}
```

```java
// Main class to instantiate and start the thread
class ThreadDemo {
    public static void main(String[] args) {
        MyThread t = new MyThread(); // Instantiation of a Thread
        t.start(); // Starting the Thread

        for (int i = 0; i < 5; i++) {
            System.out.println("main thread");
        }
    }
}
```

Thread Scheduler

- If multiple Threads are waiting to execute then which Thread will execute 1st is decided by "Thread Scheduler" which is part of JVM.
- Which algorithm or behaviour followed by Thread Scheduler we can't expect exactly it is the JVM vendor dependent hence in multithreading examples we can't expect exact execution order and exact output.

The output will be **mixed** because both threads run concurrently

```
child Thread
main thread
child Thread
main thread
child Thread
...
```

Difference between t.start() and t.run() methods.

In the case of t.start() a new Thread will be created which is responsible for the execution of run() method.

But in the case of t.run() no new Thread will be created and run() method will be executed just like a normal method by the main Thread.

In the above program if we are replacing t.start() with t.run() the Output will be sequential, not concurrent:

```
child Thread
child Thread
... (10 times)
main thread
main thread
... (5 times)
```

importance of Thread class start() method.

For every Thread the required mandatory activities like registering the Thread with Thread Scheduler will takes care by Thread class start() method and programmer is responsible just to define the job of the Thread inside run() method.

start() {

1. Register Thread with Thread Scheduler

2. All other mandatory low level activities.

3. Invoke or calling run() method. }

That is start() method acts as best assistant to the programmer. We can conclude that without executing Thread class start() method there is no chance of starting a new Thread in java. Due to this start() is considered as heart of multithreading.

If we are not overriding run() method

If we are not overriding **run()** method then Thread class run() method will be executed which has empty implementation and hence we won't get any output.

Overloading of run() method

We can overload run() method but Thread class start() method always invokes no argument **run()** method the other overload run() methods we must call explicitly then only it will be executed just like normal method.

overriding of start() method

If we override start() method then our start() method will be executed just like a normal method call and no new Thread will be started.

life cycle of the Thread

- **New/Born State**: When you **create** a thread object. The thread is in the **New state**. It is not yet started.
- **Runnable (Ready) State**: When you call ***t.start();*** The thread is registered with the **Thread Scheduler** and becomes Runnable. It is waiting for CPU allocation.
- **Running State**: When the Thread Scheduler allocates CPU, the thread moves to Running state. The **run()** method executes.
- **Dead State**: After ***run()*** completes, the thread enters the Dead state. It cannot be restarted.

```java
public class ThreadDemo {
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start(); // Valid
        t.start(); // Runtime Error: IllegalThreadStateException
    }
}
```

DEFINING A THREAD BY IMPLEMENTING RUNNABLE INTERFACE

We can define a Thread even by implementing Runnable interface also. Runnable interface present in java.lang.pkg and contains only one method run().

Java does not support multiple inheritance, so implementing Runnable is more flexible because separates task (Runnable) from thread control (Thread).

```java
// Defining a Thread by implementing Runnable
class MyRunnable implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("child Thread");
        }
    }
}
```

```java
// Main class to instantiate and start the thread
class ThreadDemo {
    public static void main(String[] args) {
        MyRunnable r = new MyRunnable(); // Target Runnable
        Thread t = new Thread(r);        // Thread object with Runnable
        t.start();                       // Start the thread

        for (int i = 0; i < 10; i++) {
            System.out.println("main thread");
        }
    }
}
```

Output:

```
child Thread
main thread
child Thread
main thread
...
```

Case Study:

```java
MyRunnable r = new MyRunnable(); // implements Runnable
Thread t1 = new Thread();        // no target Runnable
Thread t2 = new Thread(r);       // target Runnable is r
```

Case 1: t1.start()

t1 has no target Runnable, so its run() method in Thread class executes. Default run() in Thread does nothing. **Result**: A new thread starts but does nothing.

```
main thread
main thread
... (10 times)
```

Case 2: t1.run()

Calls run() like a normal method in the main thread. Since t1 has no target Runnable, run() does nothing.

Result: No new thread, no output.

```
main thread
main thread
... (10 times)
```

Case 3: t2.start()

t2 has target r (MyRunnable), so: A new **thread** is **created** and **run()** of MyRunnable executes in that new thread. Result: Prints "child Thread" 10 times (concurrently with main thread if main does other work).

```
child Thread
main thread
child Thread
main thread
... (interleaved)
```

Case 4: t2.run()

Executes **run()** of MyRunnable in the **main** thread (no new thread).

Result: Sequential execution, prints "child Thread" 10 times before main thread continues.

```
child Thread
child Thread
child Thread
main thread
main thread
... (10 times)
```

Case 5: r.start()

Compile-time error: MyRunnable does not have start() method (it's not a Thread). **Result**: Compilation fails.

```
Error: cannot find symbol method start()
```

Case 6: r.run()

No new Thread will be created and MyRunnable class **run**() method will be executed just like a normal method call..

```
child Thread
child Thread
child Thread
main thread
main thread
... (10 times)
```

Best approach to define a Thread:

Among the 2 ways of defining a Thread, implements Runnable approach is always recommended.

In the 1st approach our class should always extends Thread class there is no chance of extending any other class hence we are missing the benefits of inheritance.

But in the 2nd approach while implementing Runnable interface we can extend some other class also. Hence implements Runnable mechanism is recommended to define a Thread.

Thread class constructors

1. Thread t=new Thread();
2. Thread t=new Thread(Runnable r);
3. Thread t=new Thread(String name);
4. Thread t=new Thread(Runnable r,String name);
5. Thread t=new Thread(ThreadGroup g,String name);
6. Thread t=new Thread(ThreadGroup g,Runnable r);
7. Thread t=new Thread(ThreadGroup g,Runnable r,String name);
8. Thread t=new Thread(ThreadGroup g,Runnable r,String name, long stackSize);

Getting and setting name of a Thread

- Every Thread in java has some name it may be provided explicitly by the programmer or automatically generated by JVM.
- Thread class defines the following methods to get and set name of a Thread.
- Methods:
  - public final String getName()
  - public final void setName(String name)
- We can get current executing Thread object reference by using Thread.currentThread() method.

Thread Priorities

- Every Thread in java has some priority it may be default priority generated by JVM (or) explicitly provided by the programmer.
- The valid range of Thread priorities is 1 to 10where 1 is the least priority and 10 is highest priority.
- Thread class defines the following constants to represent some standard

- There are no constants like Thread.LOW_PRIORITY, Thread.HIGH_PRIORITY
- Thread scheduler uses these priorities while allocating CPU.
- The Thread which is having highest priority will get chance for 1st execution. If 2 Threads having the same priority then we can't expect exact execution order it depends on Thread scheduler whose behaviour is vendor dependent.
- The allowed values are 1 to 10 otherwise we will get runtime exception saying "IllegalArgumentException".
- priorities.
  - Thread. MIN_PRIORITY----------1
  - Thread. MAX_PRIORITY----------10
  - Thread. NORM_PRIORITY--------5
- We can get and set the priority of a Thread by using the following methods.
  - public final int getPriority()
  - public final void setPriority(int newPriority);//the allowed values are 1 to 10

The Methods to Prevent a Thread from Execution

We can prevent(stop) a Thread execution by using the following methods.

- yield();
- join();
- sleep();

yield();

- The yield() method in Java is a static method of the Thread class that gives a hint to the Thread Scheduler that the current thread is willing to pause and let other threads of equal priority execute.
- It does not guarantee that the current thread will stop immediately. It only suggests to the scheduler: “If there are other threads of the same priority, let them run.”
- If no other thread of equal priority is waiting, the current thread may continue.
- It does not release locks (unlike sleep() or wait()).

Join();

- The join() method in Java is used to make one thread wait for the completion of another thread before continuing. It is part of the Thread class.
- If Thread A calls join() on Thread B, then Thread A will pause until Thread B completes its execution (or the timeout expires).
- Useful when you need to ensure a thread finishes before proceeding.

Sleep();

- The sleep() method in Java is used to pause the execution of the current thread for a specified time. It is a static method in the Thread class.
- Causes the current thread to pause for the given time. Does not release locks (unlike wait()).
- Throws InterruptedException, so you must handle or declare it.
- After sleep time ends, the thread becomes Runnable again (not guaranteed to run immediately).

Interrupting a Thread

Interrupting a Thread in Java means signalling a thread that it should stop what it's doing and handle an interruption. This is done using the interrupt() method of the Thread class.

interrupt()

Sends an interrupt signal to the thread. Does not forcibly stop the thread; it sets an internal flag.

isInterrupted()

Checks if the thread has been interrupted (returns true or false).

interrupted()

Static method that checks and clears the interrupted status of the current thread.

Behaviour

- If the thread is sleeping or waiting, interrupt() causes an InterruptedException.
- If the thread is running normally, interrupt() just sets the flag; you must check it manually.

Synchronization

- Synchronized is the keyword applicable for **methods** and **blocks** but not for classes and variables.
- If a method or block declared as the synchronized then at a time only **one Thread is allowed to execute that method or block on the given object**.
- The main advantage of synchronized keyword is we can resolve date **inconsistency** problems.
- But the main disadvantage of synchronized keyword is it increases **waiting time** of the Thread and effects performance of the system.
- Internally synchronization concept is implemented by using **lock** concept.
- Every object in java has a unique lock. Whenever we are using synchronized keyword then only lock concept will come into the picture.
- If a **Thread** wants to execute any synchronized method on the given object 1st it must get the lock of that object. Once a Thread got the lock of that object then it's allowed to execute any synchronized method on that object. If the synchronized method execution completes then automatically Thread releases lock.
- While a **Thread executing any synchronized method** the **remaining** Threads are **not allowed execute any synchronized method on that object** simultaneously. But remaining Threads are **allowed to execute any non-synchronized** method simultaneously. [lock concept is implemented based on object but not based on method].

```
class Shared {
    synchronized void printNumbers() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(Thread.currentThread().getName() + " - " + i);
        }
    }
}
```

```
class MyThread extends Thread {
    Shared s;
    MyThread(Shared s) { this.s = s; }
    public void run() { s.printNumbers(); }
}
```

```
public class SyncDemo {
    Run | Debug
    public static void main(String[] args) {
        Shared obj = new Shared();
        MyThread t1 = new MyThread(obj);
        MyThread t2 = new MyThread(obj);
        t1.start();
        t2.start();
    }
}
```

Output:

```
Thread-1 - 1
Thread-1 - 2
Thread-1 - 1
Thread-1 - 2
Thread-1 - 2
Thread-1 - 3
Thread-1 - 4
Thread-1 - 5
Thread-0 - 1
Thread-0 - 2
Thread-0 - 3
Thread-0 - 4
Thread-0 - 5
```

Without synchronization the output will be random:

```
Thread-0 - 1
Thread-0 - 2
Thread-1 - 1
Thread-1 - 2
Thread-0 - 3
Thread-1 - 3
Thread-1 - 4
Thread-0 - 4
Thread-1 - 5
Thread-0 - 5
```

Dead lock

- If 2 Threads are waiting for each other forever(without end) such type of situation(infinite waiting) is called dead lock.
- There are no resolution techniques for dead lock but several prevention(avoidance) techniques are possible.
- Synchronized keyword is the cause for deadlock hence whenever we are using synchronized keyword we must take special care.

```
class DeadlockDemo {
    private final Object lock1 = new Object();
    private final Object lock2 = new Object();

    public void method1() {
        synchronized (lock1) {
            System.out.println(Thread.currentThread().getName() + " acquired lock1");
            try { Thread.sleep(millis: 100); } catch (InterruptedException e) {}
            synchronized (lock2) {
                System.out.println(Thread.currentThread().getName() + " acquired lock2");
            }
        }
    }

    public void method2() {
        synchronized (lock2) {
            System.out.println(Thread.currentThread().getName() + " acquired lock2");
            try { Thread.sleep(millis: 100); } catch (InterruptedException e) {}
            synchronized (lock1) {
                System.out.println(Thread.currentThread().getName() + " acquired lock1");
            }
        }
    }

```

```
Run | Debug
25    public static void main(String[] args) {
26        DeadlockDemo demo = new DeadlockDemo();
27        Thread t1 = new Thread(demo::method1, name: "Thread-1");
28        Thread t2 = new Thread(demo::method2, name: "Thread-2");
29        t1.start();
30        t2.start();
31    }
32  }
```

Why deadlock occurs

- Thread-1 locks lock1 then waits for lock2.
- Thread-2 locks lock2 then waits for lock1.
- Neither can proceed → deadlock.

Deadlock vs Starvation vs Livelock

Deadlock: No progress; threads blocked.

Starvation: A thread never gets CPU or resource due to unfair scheduling.

Livelock: Threads keep changing state but no progress (e.g., two polite threads keep yielding).

Daemon Threads

The Threads which are executing in the background are called daemon Threads. The main objective of daemon Threads is to provide support for non-daemon Threads like main Thread.

- It does not prevent JVM from exiting.
- JVM terminates when all user threads finish, even if daemon threads are still running.
- Common use: Garbage Collector, background monitoring tasks.

```
1   class MyDaemon extends Thread {
2       public void run() {
3           while (true) {
4               System.out.println("Daemon thread running...");
5               try { Thread.sleep(500); } catch (InterruptedException e) {}
6           }
7       }
8   }
9
```

```
10  public class DaemonDemo {
11      public static void main(String[] args) {
12          MyDaemon t = new MyDaemon();
13          t.setDaemon(true); // must be before start()
14          t.start();
15
16          System.out.println("Main thread ends...");
17          // JVM will exit after main finishes because only daemon thread remains
18      }
19  }
```

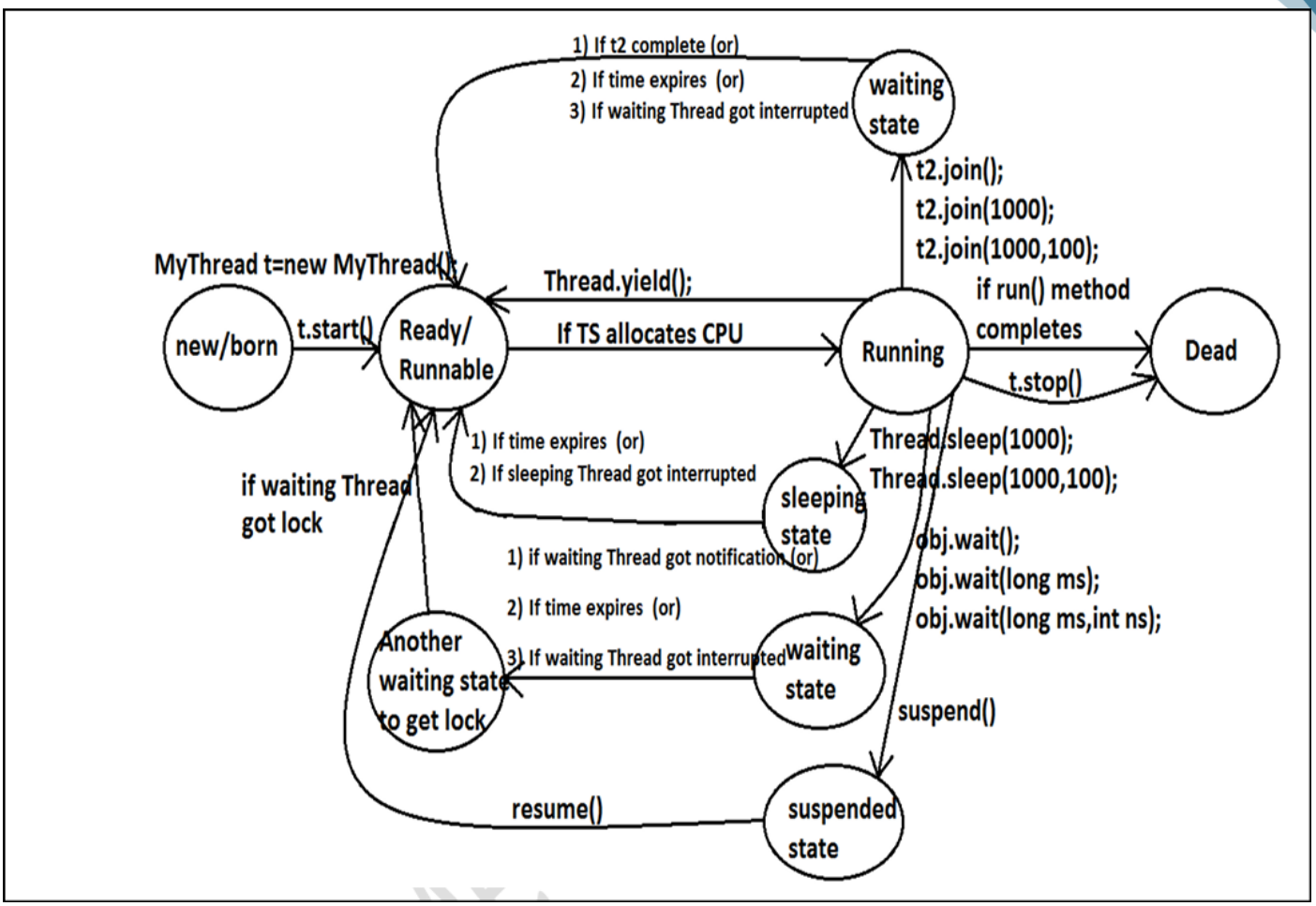
Lazy Thread

- A thread that is not created or started immediately, but only when its work is required. This is a lazy initialization pattern applied to threads.
- Common in resource-sensitive systems where you avoid creating threads until absolutely necessary.

RACE condition

- Executing multiple Threads simultaneously and causing data inconsistency problems is nothing but Race condition
- we can resolve race condition by using synchronized keyword.

Life cycle of a Thread



Thread States in Java

1.New/Born

- Thread object created:

```
1  MyThread t = new MyThread();
```

- Not yet started.

2.Ready / Runnable

- After calling t.start(), thread enters Runnable state.
- Waiting for CPU allocation by Thread Scheduler.

3.Running

- When CPU is allocated by the scheduler.
- Can call Thread.yield() to give up CPU voluntarily.

4.Sleeping State

- When Thread.sleep(ms) is called.
- Transitions back to Runnable after time expires or interrupted.

5.Waiting State

- When join(), wait(), or timed wait methods are called
  - t2.join()
  - obj.wait()

- Comes back to Runnable when
  - Time expires
  - Notification received (notify() / notifyAll())
  - Interrupted

6.Suspended State (Deprecated in modern Java)

- Caused by suspend() (unsafe, deprecated).
- Resumes with resume().

7.Dead

- When run() method completes or stop() is called (also deprecated).